S-PLUS 7
Enterprise Developer
User's Guide

April 2005

Insightful Corporation
Seattle, Washington

Proprietary Notice

Insightful Corporation owns both this software program and its documentation. Both the program and documentation are copyrighted with all rights reserved by Insightful Corporation.

The correct bibliographical reference for this document is as follows:

S-PLUS 7 Enterprise Developer User's Guide, Insightful Corporation, Seattle, WA.

Printed in the United States.

Copyright Notice

Copyright © 1987-2005, Insightful Corporation. All rights reserved.

Insightful Corporation
1700 Westlake Avenue N, Suite 500
Seattle, WA 98109-3044
USA

Trademarks

Insightful, Insightful Corporation, the Insightful logo, S-PLUS, Insightful Miner, S+FinMetrics, S+SeqTrial, S+SpatialStats, S+ArrayAnalyzer, S+EnvironmentalStats, S+Wavelets, S-PLUS Graphlets and Graphlet are either trademarks or registered trademarks of Insightful Corporation in the United States and/or other countries. Intel and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Microsoft, Windows, MS-DOS and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All product names mentioned herein may be trademarks or registered trademarks of their respective companies.

ACKNOWLEDGMENTS

S-PLUS would not exist without the pioneering research of the Bell Labs S team at AT&T (now Lucent Technologies): John Chambers, Richard A. Becker (now at AT&T Laboratories), Allan R. Wilks (now at AT&T Laboratories), Duncan Temple Lang, and their colleagues in the statistics research departments at Lucent: William S. Cleveland, Trevor Hastie (now at Stanford University), Linda Clark, Anne Freeny, Eric Grosse, David James, José Pinheiro, Daryl Pregibon, and Ming Shyu.

Insightful Corporation thanks the following individuals for their contributions to this and earlier releases of S-PLUS: Douglas M. Bates, Leo Breiman, Dan Carr, Steve Dubnoff, Don Edwards, Jerome Friedman, Kevin Goodman, Perry Haaland, David Hardesty, Frank Harrell, Richard Heiberger, Mia Hubert, Richard Jones, Jennifer Lasecki, W.Q. Meeker, Adrian Raftery, Brian Ripley, Peter Rousseeuw, J.D. Spurrier, Anja Struyf, Terry Therneau, Rob Tibshirani, Katrien Van Driessen, William Venables, and Judy Zeh.

CONTENTS

Acknowledgments iii

Chapter 1 Introduction 1

Welcome to the S-PLUS Enterprise Developer User's Guide 2

Analyzing Large Data Sets 4

Advanced Programming 8

Chapter 2 The S-PLUS Workbench 9

Introduction 11

Starting the S-PLUS Workbench 16

S-PLUS Perspective 24

Views 26

Script Editor 38

S-PLUS Workbench Tasks 41

Commonly-Used Features in Eclipse 56

Chapter 3 The Big Data Library 59

Introduction 60

Working with a Large Data Set 61

Size Considerations 65

The Big Data Library Architecture 69

Chapter 4 Exploring and Manipulating Large Data Sets **85**

Introduction 86

Working in the S-PLUS Environment 87

Manipulating Data: Census Example 91

Manipulating Data: Stock Sample 105

Chapter 5 Creating Graphical Displays of Large Data Sets **115**

Introduction 116

Overview of Graph Functions 117

Example Graphs 123

Chapter 6 Modeling Large Data Sets **159**

Introduction 160

Overview of Modeling 161

Building a Model 162

Predicting from the Model 180

Chapter 7 Advanced Programming Information **185**

Introduction 186

Big Data Block Size Issues 187

Big Data String and Factor Issues 191

Storing and Retrieving Large S Objects 197

Increasing Efficiency 199

Appendix: Big Data Library Functions **201**

Introduction 202

Big Data Library Functions 203

INTRODUCTION 1

Welcome to the S-PLUS Enterprise Developer User's Guide 2

Analyzing Large Data Sets 4
- Out-of-Memory Data Storage 4
- Big Data Library Options in the S-PLUS Environment 4
- Working with Large Data Sets 5

Advanced Programming 8
- More Advanced Programming Concepts and Tasks 8

WELCOME TO THE S-PLUS ENTERPRISE DEVELOPER USER'S GUIDE

The Big Data library is a significant addition to the S-PLUS family of libraries. It provides objects, classes, and functions to manipulate, model, and explore large data sets using the S language.

S-PLUS Enterprise Developer includes the S-PLUS Workbench, the S-PLUS customization of the Eclipse Integrated Development Environment. It also includes the premier data analysis package and the ability to handle both small and large data sets. Programmers familiar with the S language will be comfortable immediately with the Big Data library's object-oriented design and syntax. It is designed to work with existing S-PLUS functions, and many functions available in the S-PLUS engine also work with large data sets. Conversely, Big Data library functions work with small data sets. For a comprehensive list of the Big Data library functions, see the Appendix...

Note

The Big Data library loads by default only in the Windows S-PLUS GUI and from S-PLUS BATCH. The Big Data library is not loaded by default when you start S-PLUS from the S-PLUS Workbench, the Unix or Windows Command line, or as a Console application. Always load the Big Data library if you work with big data projects. (If you start a big data project without having loaded the Big Data library, you will see errors when you run your script.)

To set the option to start S-PLUS without loading the Big Data library, in the Windows S-PLUS GUI, on the menu, click **Options ► General Settings**, and then click the **Startup** tab. Clear the **Load Bigdata library** check box.

Note

When you work with large data sets using S-PLUS, use the **Commands** window and **Script** window. See the S-PLUS *User's Guide* for more information on these user interfaces. there are no equivalent GUI functions available in this release.

Using S-PLUS and the Big Data library, you can:

- Import a large data set from a text file or a database.
- Convert data frames to big data objects and vice versa.

- Manage projects and code files in the S-PLUS Workbench.
- View data in the **Data Viewer**.
- Split or append a data set.
- Clean, sort, and filter rows and columns in a data set.
- Create plots using hexbin plotting.
- Fit models to large data sets.
- Export the large data set.
- Create your own functions that use large data sets.

ANALYZING LARGE DATA SETS

This section includes:

- The architecture of the Big Data library.
- A description of the options in the S-PLUS environment for working with large data sets.
- An outline of the tasks associated with importing, manipulating, modeling, and plotting large data sets, mapped to the procedures in the outline of this manual.

Out-of-Memory Data Storage

The S language was originally designed to store data objects in memory to provide the fastest data analysis possible. For example, when you create a data frame object, as follows:

```
mydata <- read.table("datafile.txt")
```

all of the data in the object `mydata` is manipulated in random-access memory (RAM). If your computer does not have enough RAM to hold the data, your computer returns an out-of-memory message. S-PLUS Enterprise Developer includes the Big Data library, which provides functions to store and manipulate data out-of-memory. For a more in-depth discussion about how the Big Data library uses out-of-memory data storage to help solve this problem, see Chapter 3, The Big Data Library.

Big Data Library Options in the S-PLUS Environment

The S-PLUS graphical user interface (GUI) in Microsoft Windows and the S-PLUS Workbench in Microsoft Windows or Unix platforms provide limited support for working with large data frames. You can use the S-PLUS GUI in Microsoft Windows to import, export, and view data in the **Data Viewer**. Otherwise, you must call Big Data Library functions by typing them at the prompt in the **Commands** window. For more information about importing or exporting large data sets, see Chapter 3, The Big Data Library.

Working with Large Data Sets

When you work with a large data set, you can perform any or all of the tasks illustrated in Figure 1.1.

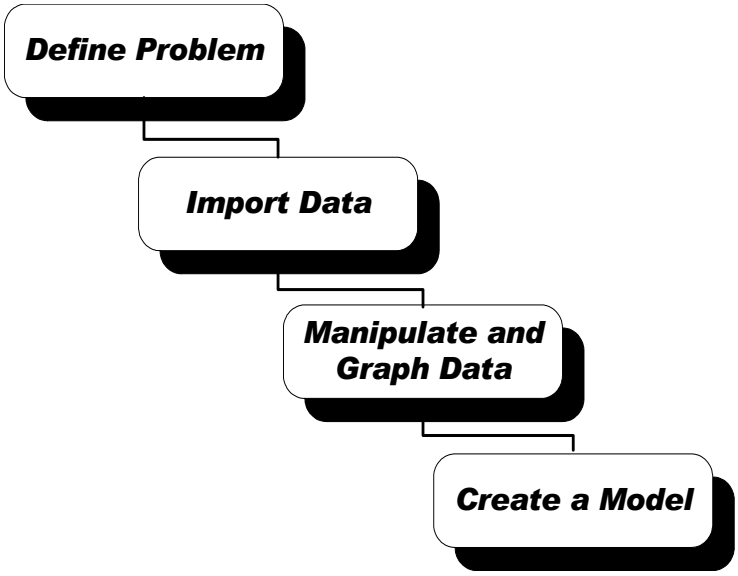


Figure 1.1: *Big Data tasks*

You might just be importing or manipulating data, or building a graphical display, or modeling data. This section outlines these high-level tasks, first discussing the concepts behind defining your data problem, and then dividing each high-level task into procedures to accomplish the tasks.

Define the Problem

For nearly every investigation, understanding the problem and planning for its solution can save you time, energy, and money later. Defining the problem is key to determining the type of information you want to derive from your data and the best strategy for extracting the information and demonstrating the answer.

- **Determine the question you are trying to answer**. What is the objective of your inquiry?
- **Identify the variables in your data that can answer the question**. Your data set might contain much more information than you need to determine the answer to your question, and it also might include blank fields or errors. You must filter and remove factors that are not essential to your answer.

- **Design the model**. You can create a model that predicts behavior.

Import the Data

Using S-PLUS and the Big Data library, you can import data from the source types listed in Table 5.1, in Chapter 5, Importing and Exporting Data in the S-PLUS *User's Guide*.

The easiest way to import a large data set is by typing the `importData` command directly in the S-PLUS **Commands** Window or the **Console View** in the S-PLUS Workbench, specifying the argument `bigdata=T`. For more information about using the **Commands** window, see the S-PLUS *User's Guide*. For more information about `importData`, see its help topic. For more information about deciding when to set `bigdata=T`, see the section When to Set `bigdata=T` on page 65.

Alternatively, in Microsoft Windows, import data using the **Import Data** dialog box in the S-PLUS GUI. For more information about using the **Import Data** dialog box in Windows, click **Help ▶ Available Help ▶ S-PLUS Help**, and then see the topic Importing Data Files.

Manipulate the Data

Once your data is imported into S-PLUS, you can view and manipulate the data. Chapter 4, Exploring and Manipulating Large Data Sets contains more in-depth discussions of the following data manipulation tasks:

- Converting data.
- Generating a large data vector.
- Displaying data as a big data frame (`bdFrame`).
- Exploring data.
- Cleaning data.
- Splitting data.
- Appending data sets.
- Manipulating and filtering rows and columns.
- Manipulating time series objects.
- Exporting data.

Build a Graphical Display

Once your data has been cleaned, sorted, and filtered in S-PLUS, you can optionally build a graphical display as an initial step towards assessing trends in your data. contains more in-depth discussions of the following graphics tasks:

- Plotting using hexagonal bins.
- Creating a traditional graph.
- Creating a Trellis graph.
- Evaluating and aggregating data over a grid using traditional or Trellis graphs.
- Creating time series graphs.

Create a Model

After you have examined an initial graph of your data, you can decide how you plan to model the data. The Big Data library contains support for the following model types:

- Linear Regression Model
- Generalized Linear Model
- Principal Components
- Clustering

Chapter 6, Modeling Large Data Sets contains more in-depth discussions of the following tasks:

- Building a Model
- Predicting from Small Data Models

ADVANCED PROGRAMMING

Whether you are new to S-PLUS or an experienced user, consider taking advantage of the more advanced features available with the Big Data library.

More Advanced Programming Concepts and Tasks

Chapter 7, Advanced Programming Information, discusses more complicated issues, including:

- Enhancing performance.
- Splitting and aggregating data.
- Performing by-block computations.
- Creating your own Big Data subclasses.
- Writing your own Big Data functions, classes, and methods.
- Writing general computations with `bd.block.apply`.

THE S-PLUS WORKBENCH 2

Introduction **11**

Starting the S-PLUS Workbench **16**
- S-PLUS Workspace 16
- S-PLUS Preferences 18
- New Project Wizard 23

S-PLUS Perspective **24**
- Changing the S-PLUS Workbench Perspective 24

Views **26**
- Customizing the Perspective's Views 27
- S-PLUS Workbench Console View 28
- History View 29
- Objects View 30
- Outline View 31
- Output View 33
- Problems View 34
- Search Path View 35
- Tasks View 36

Script Editor **38**
- Text Editing Assistance 38
- View integration 39
- Menu Options 39

S-PLUS Workbench Tasks **41**
- Creating a Project 41
- Setting the Project's Preferences 44
- Customizing the S-PLUS Workbench Default Perspective and Views 45
- Changing Attached Databases 46
- Creating a Script 48

Editing Code in the Script Editor 49
Running Code 52
Fixing Problems in the Code 54
Closing the Project 54

Commonly-Used Features in Eclipse 56

INTRODUCTION

S-PLUS provides a plug-in, or customization, of the Eclipse Integrated Development Environment (IDE) called the S-PLUS Workbench. You can use the S-PLUS Workbench and the basic Eclipse IDE features to manage your project files, provide source control for shared project files, edit your code, run S-PLUS commands, and troubleshoot problems with S-PLUS projects.

The S-PLUS Workbench is a stand-alone application that runs the S-PLUS engine. When you run the S-PLUS Workbench, you do not need to run any other version of S-PLUS (for example, the console or traditional Windows or Java GUI).

Caution

If you run two or more simultaneous sessions of S-PLUS (including one or more in the S-PLUS Workbench), take care to use different working directories. To use the same working directory for multiple sessions can cause conflicts, and possibly even data corruption.

This chapter contains descriptions of the features and a task-centered tutorial for the S-PLUS implementation of Eclipse (the S-PLUS Workbench). Before you begin using the S-PLUS Workbench, you should understand key terms and concepts that vary from the traditional S-PLUS Windows GUI and Java GUI.

The Eclipse IDE contains extensive, in-depth documentation for its user interface. For information about basic Eclipse IDE functionality, see the integrated documentation, the *Workbench User Guide*.

Note

If you are using the Eclipse IDE on a Unix platform from a Windows machine using a Windows X-server software package, you might notice that Eclipse runs slowly, similar to the S-PLUS Java GUI. See the Release Notes for more information and recommendations for improving UI performance.

Table 2.1: *Important terms and concepts.*

Term	Definition
Perspective	Defines the preferences, settings, and views for working with Eclipse projects. The S-PLUS perspective is conceptually equivalent to the traditional S-PLUS Windows GUI or Java GUI. Use the S-PLUS perspective as the primary perspective for interactive command line use of S-PLUS. For an example of changing the perspective, see the section Customizing the S-PLUS Workbench Default Perspective and Views on page 45.
Workspace	A physical directory on your machine that manages S-PLUS Workbench resources such as projects and other options. On your machine's hard drive, the Workspace directory contains the S-PLUS .Data database and the Eclipse .metadata database. (You should never touch these resources.) Notice that the .Data database is associated with the Workspace, rather than the S-PLUS Project. This design is different from the association you notice when you work in S-PLUS in its other environments. When you start the S-PLUS Workbench, you are prompted to create or identify the Workspace. See the section S-PLUS Workspace on page 16.

Table 2.1: *Important terms and concepts. (Continued)*

Term	Definition
Project	A resource containing text files, scripts, and associated files. The S-PLUS Workbench project is used for build and version management, sharing, and resource management. Before you begin working with any files in the S-PLUS Workbench, you must create a project. You can: • Create an empty new directory located in your specified Workspace directory, and then either create a new script or import an existing project directory (i.e., copy the files). • Select an existing directory containing project files at an alternate location (i.e., work with the files at the specified location). See the section Creating a Project on page 41.
View	Integrated windows, containing their own menus and commands, that display specific parts of your data and projects and provide tools for data manipulation. Includes the **Console View**, **History View**, **Objects View**, **Outline View**, **Output View**, **Problems View**, **Search Path View**, and **Tasks View**. For practice exercises working with views, see the section S-PLUS Workbench Tasks on page 41.
Editor	An integrated code/text editor that includes support for syntax coloring, text formatting, and integration with the other views. Analogous to the Script Editor in the traditional S-PLUS GUI. See Script Editor. To practice using the Script Editor, see the section Editing Code in the Script Editor on page 49.

Getting Started Tutorial

If you are not familiar with the Eclipse IDE, once you start the S-PLUS Workbench, take the first few minutes to learn the basic concepts and IDE layout by working through the basic tutorial in the *Workbench User Guide*.

To View the Eclipse Getting Started Tutorial

1. From the Workbench main menu, click **Help ▶ Help Contents**.
2. In the right pane, expand the table of contents by clicking **Workbench User Guide**.
3. Click **Getting Started**, and then click **Basic tutorial**.

The *Workbench User Guide* opens in a separate window; you can toggle back and forth between the Workbench application and the User Guide.

Additional S-PLUS Customizations

The S-PLUS Workbench includes the following additional customizations to the basic Eclipse UI:

- **Customized Menus**: The S-PLUS Workbench provides customizations to Eclipse menu options. For more information, see the section Menu Options on page 39.
- **Function Help**: The S-PLUS Workbench provides access to function help topics.
  - In the **Console View**, type `help(`*`functionname`*`)` where *`functionname`* is the function for which you want help.
  - In the Script Editor, highlight the function for which you want help, and then type F1.
  - Use the S-PLUS Workbench menu options. In the Script Editor, select the function for which you want help, and then, on the menu click either:
    - **Source ▶ Open S-PLUS Help File**
    - **Help ▶ S-PLUS Help**

Note

If you click either menu option with no function selected in the Script Editor, the S-PLUS Workbench displays the `help` function topic.

- **Script Running Options**: The S-PLUS Workbench provides the following customized solutions for running your scripts.

- **Copy to Console**: Available from the right-click menu in the Script Editor, this option copies the selected code and pastes it into the **Console View**. See the section Copying Script Code to the Console on page 52.
- **Run**: Available by pressing F9, or on the toolbar, from the right-click menu in the Script Editor, this option runs the selected code (or all code, if none is selected), and displays output in the **Output View**. See the section Running Code and Reviewing the Output on page 54 for more information.

S-PLUS Features Changed and Eclipse Features Not Supported by the S-PLUS Workbench

- In the traditional S-PLUS GUI, you use F10 to run code. Eclipse reserves F10 to switch focus to the main menu; therefore, the S-PLUS Workbench specifies F9 to run code.
- The S-PLUS Workbench does not implement the Eclipse **Run** menu item. Selecting this menu option does nothing.
- The S-PLUS Workbench does not support Eclipse's **Project ▶ Build** menu items.
- Currently, the S-PLUS Workbench does not support Eclipse's **Debug** perspective. To debug S-PLUS Scripts, in the Script Editor, use the S-PLUS debugging functions, such as `inspect`, `browser`, `debugger`, and others. For more information, see Chapter 7, Debugging Your Functions, in the *S-PLUS Programmer's Guide*.

STARTING THE S-PLUS WORKBENCH

The S-PLUS Workbench user interface is the same in both Microsoft Windows and Unix platforms.

From Microsoft Windows

In Microsoft Windows, click the **Start** menu ▶ **All Programs** ▶ **S-PLUS 7.0** ▶ **S-PLUS Workbench**.

From Unix

In Unix, at the command prompt, type

```
Splus -w
```

or type

```
Splus -workbench
```

S-PLUS Workspace

The S-PLUS Workspace is the directory where the S-PLUS Workspace .Data and Eclipse .metadata databases are stored. (You should never touch these files.) Optionally, the Workspace directory can also store your project directories. The S-PLUS Workspace is the default directory specified for the project's directory in the **New Project** wizard. See the section New Project Wizard on page 23 for more information.

The S-PLUS Workspace .Data directory is associated with the Workspace, not individual projects. That is, the Workspace .Data directory stores all objects for all project directories associated with

the Workspace. This design varies from the traditional S-PLUS project design, where the .Data directory is associated with a single project and contains objects only for that project..

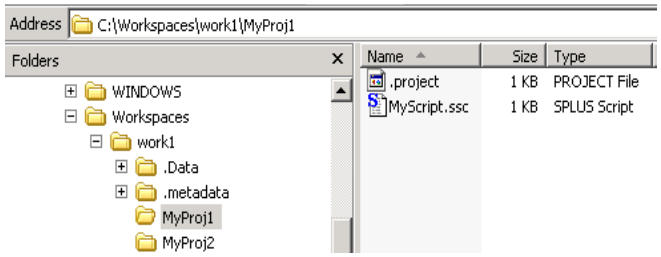


Figure 2.1: *Workspace directory showing* ***.Data*** *directory,* ***.metadata*** *directory, and project directories*

Important

By default, the S-PLUS Workbench reads objects from the *Workspace*'s .Data directory, while traditional S-PLUS reads objects from the *project's* .Data directory. Therefore, if you create a project using the S-PLUS Workbench, and then open that project in the traditional S-PLUS GUI, you must attach the Workspace's .Data directory to see its objects. The reverse is also true: If you open a project in the S-PLUS Workbench that you have previously opened in the traditional S-PLUS GUI, you must attach the project's .Data directory to see its objects. (By default, the traditional S-PLUS 7 working directory is **C:\Program Files\Insightful\splus70\users*username*\.Data**.)

When working with S-PLUS Workbench projects:

- Never store your project files directly in your Workspace directory. (Project files--including the **.project** file--should be in their own directory.)
- Avoid nesting projects (that is, create one project in a subdirectory of another project).

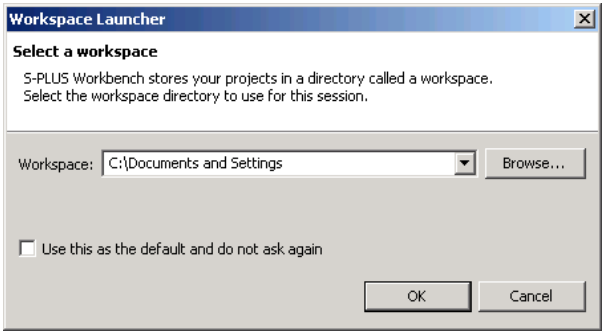

Workspace Launcher
Select a workspace
S-PLUS Workbench stores your projects in a directory called a workspace.
Select the workspace directory to use for this session.
Workspace:
C:\Documents and Settings
Browse...
Use this as the default and do not ask again
OK
Cancel

Figure 2.2: *Workspace Launcher dialog box*

Setting the Workspace

When you first launch the S-PLUS Workbench, you are prompted to supply the path to your S-PLUS Workspace.

To Set the Workspace

1. In the **Workspace Launcher** dialog box, specify the directory location where the Workspace **.Data** and **.metadata** databases will be stored.
2. Indicate whether you want to be prompted in future sessions to identify a Workspace using this dialog box.

Changing the Workspace

You can switch to another Workspace from within the S-PLUS Workbench user interface.

To Open a Different Workspace in S-PLUS Workbench

1. Save your work.
1. Click **File ► Switch Workspace**.
2. In the **Workspace Launcher** dialog box, provide the new Workspace location.

Note
When you switch workspaces during an S-Plus Workbench session, the current session closes, and a new session of S-Plus Workbench starts, using the new Workspace location.

S-PLUS Preferences

When you open the S-PLUS Workbench, the IDE defaults are set to the default S-PLUS perspective. The default perspective preferences include project type, window appearance, editor preferences, menu options, and file associations.

You can change these preferences, and any other default Eclipse preferences in the **Preferences** dialog box. It is available from the **Window** menu.

On the menu, click **Windows ► Preferences**. For more information about setting preferences, see the *Eclipse Workshop User's Guide*.

The S-PLUS Workbench sets default preferences in the following areas:

- **File Associations**: S-PLUS recognized file types include *.q, *.ssc, and *.t. Any of these files, which are associated with the S-PLUS Script editor, are checked for syntax errors and scanned for task tags. (S-PLUS also recognizes plain text, or *.txt, files.)

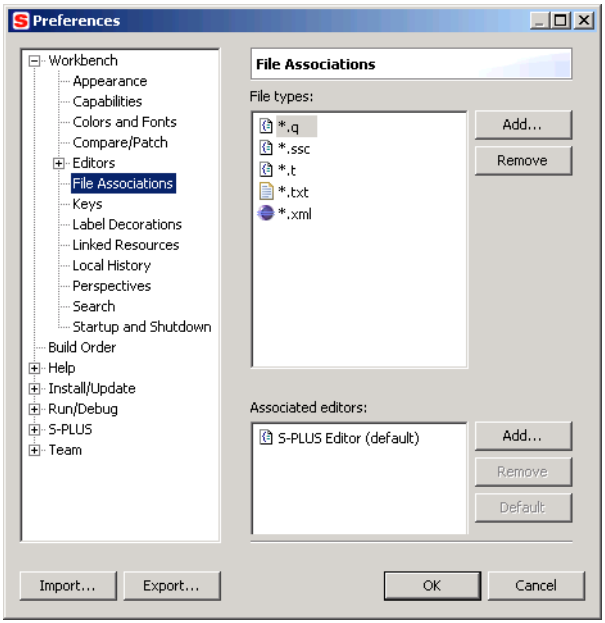


Figure 2.3: *S-PLUS **File Associations** dialog box*

- **S-PLUS Console Options**: These options control settings for the S-PLUS Workbench **Console View**.
  - **Background Color:** By default, the S-PLUS **Console View** uses the system default. Select **Custom Color**, and then click the color button to display the **Color** dialog box and choose a different background color.

- **Choose Input Color / Choose Output Color**: By default, the **Console View** displays input and output as blue and red respectively. You can select a custom color by clicking the color button, and then, in the **Color** dialog box, select a color for the input or output.

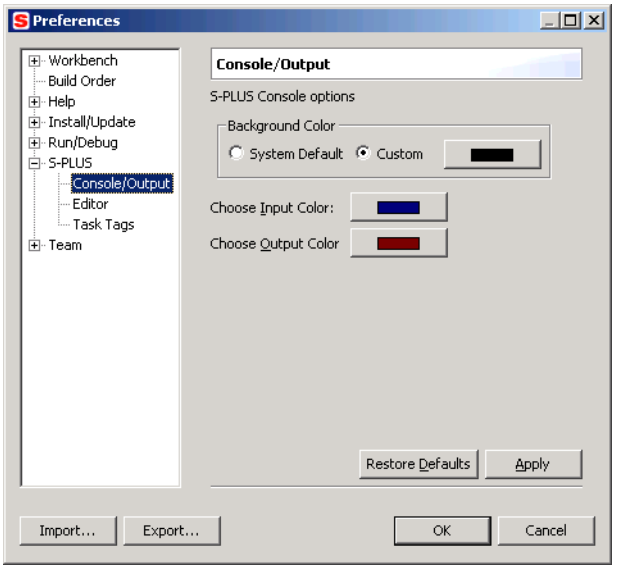


Figure 2.4: *S-PLUS **Console Options** dialog box*

- **S-PLUS Workbench options**: These options control general settings for the S-PLUS Workbench.
  - **Run code on startup**: Select this option, and then provide any code that you want the S-PLUS Workbench to run when it starts up. Note that this box is cleared by default, so no additional libraries (including the Big Data library) are loaded by default.

Note
If you clear the **Run code on startup** box, or if you remove the option to load the Big Data library on startup, and then later open a project that uses the bigdata library, you could see unexpected results when you try to perform actions. If your projects typically include large data sets, then select this option to always load the bigdata library when you start the S-PLUS Workbench.

- **Show Anonymous Functions in Outline**: By default, the S-PLUS Script editor shows anonymous functions in the outline.
- **Functions to Watch**: Contains a predefined list of S-PLUS functions to identify in the **Outline View**. You can add your own functions to this list using the **New** button. You can also remove functions from the list or reorder the list.

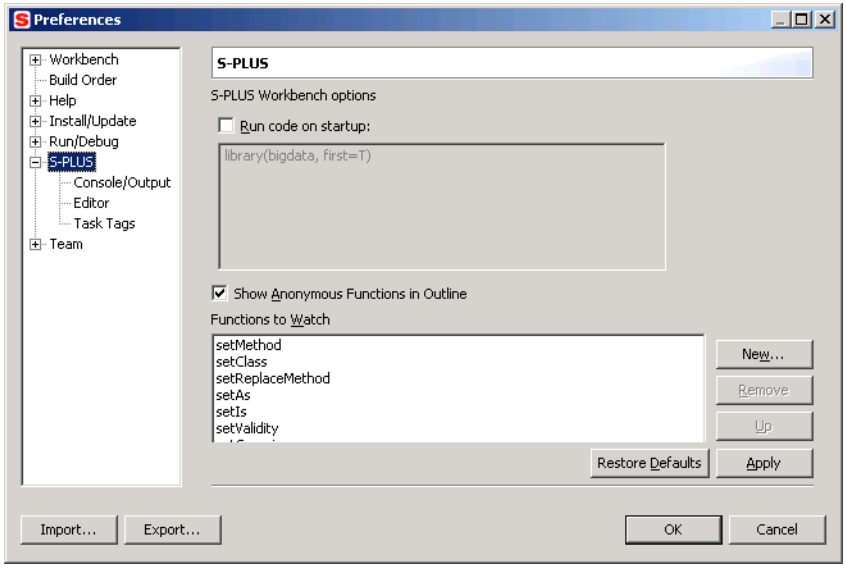


Figure 2.5: *S-PLUS **Workbench Options** dialog box*

- S-PLUS **Editor Options**: These options control settings for the S-PLUS Workbench Script Editor.
  - **Show Line Numbers:** By default, the S-PLUS Script editor shows line numbers.
  - **Background Color**: By default, the S-PLUS Script Editor uses the system background color. Select **Custom Color**, and then click the color button to display the **Color** dialog box and choose a different background color.

- **Foreground**: You can select a custom color for each of the text types listed in the **Foreground** box by selecting the text type, and then clicking Choose Color. Select the color for the text type from the **Color** dialog box.

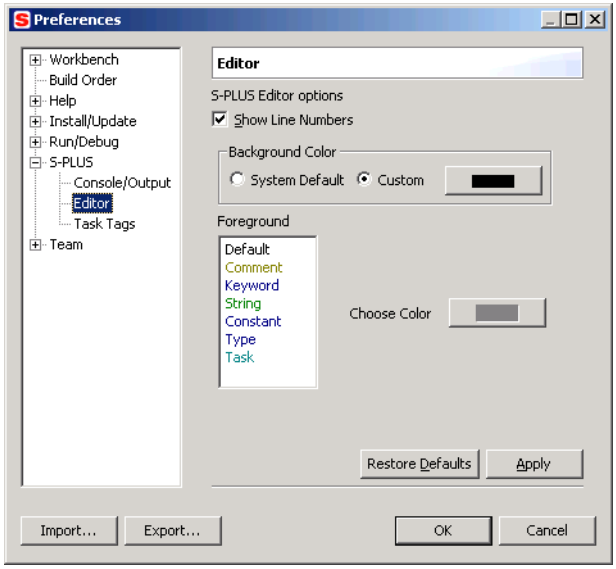


Figure 2.6: *S-PLUS **Editor Options** dialog box*

- **Task Options**: Lists the three pre-defined default task tags. See the section Tasks View on page 36 for more information.

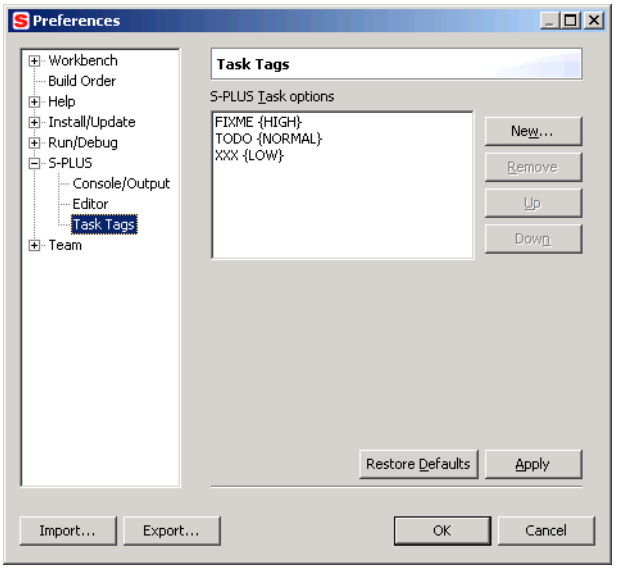


Figure 2.7: *S-PLUS **Task Options** dialog box*

New Project Wizard

When you start a new S-PLUS project in the S-PLUS Workbench, you see the **New Project** wizard, where you specify the location of your project files. See the section Creating a Project on page 41 for more information about specifying the project file location.

Working with Files External to the Project

You can use the Eclipse editor to edit non-project files in the S-PLUS Workbench. To open a non-project file, on the **File** menu, click **Open External File**, and then browse to the location of the file to edit. For more information about editing files in Eclipse, see the *Eclipse User's Guide.*

S-PLUS PERSPECTIVE

The perspective provides functionality aimed at accomplishing specific types of tasks or working with specific types of resources. The S-PLUS Workbench perspective defines the appearance and behavior for using S-PLUS, including the editor, views, menus, and toolbars.

Changing the S-PLUS Workbench Perspective

You can change the perspective to suit your development style by moving, hiding, or closing views. For more information about customizing the views within the perspective, see the section Customizing the Perspective's Views on page 27. For practice exercises customizing the perspective, see the section Customizing the S-PLUS Workbench Default Perspective and Views on page 45.

- To customize the default S-PLUS perspective, on the menu, click **Window ► Customize Perspective**. The **Customize Perspective** dialog box has two pages: **Shortcuts** and **Commands**. Each of these pages describes global changes you can make to the perspective.
- To save a changed perspective, click **Window ► Save Perspective As**.
- To restore an unsaved perspective's default settings, click **Window ► Reset Perspective**.
- To open another perspective, click **Window ► Open Perspective**, and then select a perspective from the **Select Perspective** dialog box.

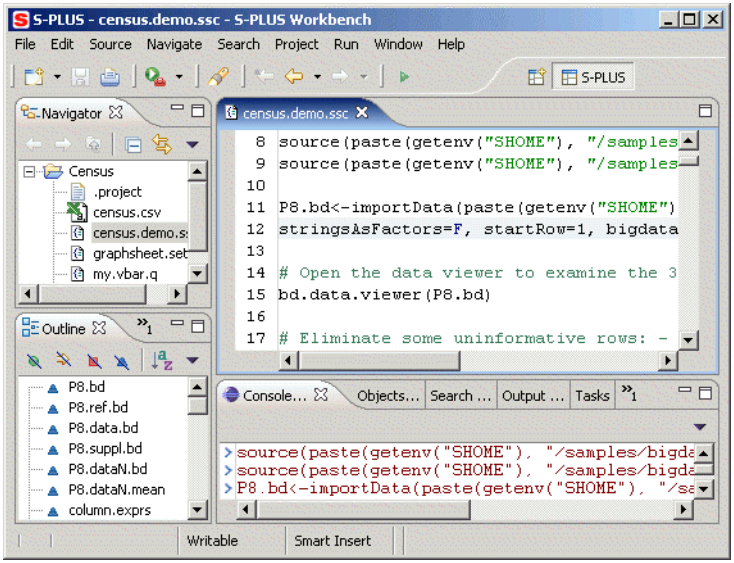


Figure 2.8: *S-PLUS Workbench window*

VIEWS

A view is a visual component in the workbench. Views support the script editor by providing alternate means of navigating through, working with, and examining the elements of the project. All views except the **Outline View** feature their own context (right-click) menus, with menu items that act on the type of data displayed in the view.

Each view contains a control menu listing actions that apply specifically to the view. The control menu is displayed either when you click the drop-down button ( ), located in the upper right corner of each view, or when you right click the view. Each view action also has a quick-key sequence to perform an action (For example, to clear the text in the console, with the **Console View** active, type CTRL+L.)

When you modify an item in a view, it is saved immediately. Normally, only one instance of a particular type of view can exist in the Workbench window. Customized views in the S-PLUS Workbench include the following:

Table 2.2: *S-PLUS Workbench views and exercise references*

View	Practice exercise
Console View	See the exercise in the section To Run Copied Script Code on page 53.
History View	See the exercise in the section To Examine the History on page 53.
Objects View	See the exercise in the section To Examine the Objects on page 51.
Outline View	See the exercise in the section To Examine the Outline on page 50.
Output View	See the exercise in the section To Run Code on page 54.

Table 2.2: *S-PLUS Workbench views and exercise references*

View	Practice exercise
Problems View	See the exercise in the section To Examine Problems on page 54.
Search Path View	See the exercise in the section Adding a Database on page 46 and section Detaching a Database on page 47.
Tasks View	See the exercise in the section To Add a Task in the Script File on page 52 and section To Add a Task Directly to the Tasks View on page 51.

These views are discussed in the following sections, and corresponding exercises for using the views are listed above. S-PLUS also uses the default **Navigator View**, which displays project directories and all files associated with the project. The Navigator View The Eclipse IDE contains other views described in the Eclipse *Workbench User's Guide.*

Customizing the Perspective's Views

The default perspective settings control the views that open by default in preset locations in the Workbench UI; however, you can customize the view appearance, and then save the resulting perspective. See the section Customizing the S-PLUS Workbench Default Perspective and Views on page 45 for more information.

Using the standard Eclipse IDE features, you can:

- Close a view by clicking the **X** icon on the view tab.
- Reposition a view by clicking its tab and dragging it to another part of the UI.
- Set a selected view to "Fast View." This option hides the view to free space in the **Workbench** window and places a minimized icon, which you can click to open the view, on the shortcut bar.
- Change the views you see in the perspective. See the section To Change the Displayed Views on page 46.

S-PLUS Workbench Console View

The S-PLUS Workbench **Console View** is an editable view, analogous to the **Commands** window in the S-PLUS GUI. Using the **Console View**, you can:

- Run individual S-PLUS commands by typing them and pressing ENTER.
- Copy an individual command or blocks of commands from the script editor, using the **Copy to Console** menu item, to run them in the **Console View**. (Note that you do not need to select **Paste**; **Copy to Console** copies your selected text in the Script Editor and pastes it into the **Console View**.)

You can use the **Console View** control menu (click ▾) to perform the following tasks:

- Clear the contents of the console.
- Copy the selected text.
- Cut the selected text.
- Paste text from the clipboard to the console.
- Find a string.
- Select all text.
- Save the console contents to a file.
- Print the console contents.

For exercises using the S-PLUS Workbench **Console View**, see the section Copying Script Code to the Console on page 52. For more information about the S-PLUS **Commands** window, see Chapter 10, Using the Commands Window in the *S-PLUS User's Guide*.

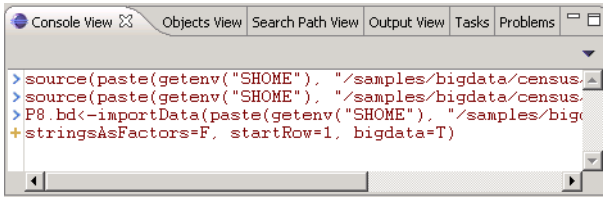


Figure 2.9: *S-PLUS Workbench Console view*

Note

If your script contains a command to open a graph or the data viewer, these windows launch externally to the S-PLUS Workbench. Note that these windows open separate from the S-PLUS Workbench, so multiple instances launch without focus, hidden behind the S-PLUS Workbench window.

History View

The **History View** is similar to the **Commands History** dialog box in S-PLUS for Windows. The **History View** is a scrollable list of commands that have previously been run in the **Console View**. (Commands that you run by clicking **Run** or pressing F9 do not appear in the **History View**. See the section Output View on page 33.)

- When you select a command in the **History View**, the pending text in the **Console View** changes to the selected text. You can then press ENTER, or you can double-click the text in the **History View** to execute the command. You can select only one line at a time in the **History View**.
- When you scroll up or down through previously-run commands in the **Console View**, the corresponding command is highlighted in the **History View**.

Note

While S-PLUS uses the key F10 to run a selected command, the S-PLUS Workbench uses the key F9 to run a selected command.

You can use the **History View** control menu (click ▾) to select input displayed in the **History View** and copy it to the **Console View**.

By default, the **History View** holds up to 150,000 lines of commands

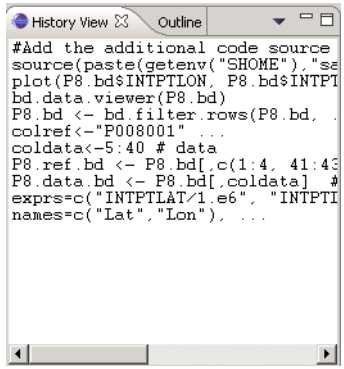


Figure 2.10: *S-PLUS Workbench History view*

Objects View

The **Objects View** is similar to the Object Explorer in the S-PLUS GUI. It displays all objects for projects associated with the Workspace. (See the section S-PLUS Workspace on page 16 for more information about the Workspace **.Data** database.) The S-PLUS Workbench **Objects View** also provides a list of the names and types of objects in S-PLUS databases. The **Objects View** includes the following information about each object:

- name
- data class
- storage mode
- extent
- size
- creation or change date.

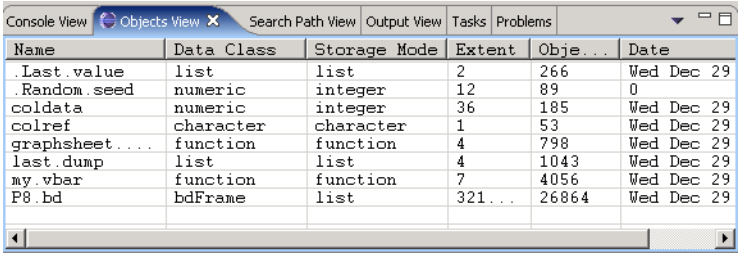
You can use the **Objects View** control menu (click ▾) to perform the following tasks:

- Select another database.
- Refresh the view on the currently-active database.
- Remove the selected object from the currently-active database.

•

Note

When you run code that creates objects in an S-PLUS script, the **Objects View** is not automatically refreshed to display the new objects. To refresh **Objects View** and display newly-created objects, right-click the **Objects View** (or click the control menu button), and then from the menu, click **Refresh**.



Name	Data Class	Storage Mode	Extent	Obje...	Date
.Last.value	list	list	2	266	Wed Dec 29
.Random.seed	numeric	integer	12	89	0
coldata	numeric	integer	36	185	Wed Dec 29
colref	character	character	1	53	Wed Dec 29
graphsheet....	function	function	4	798	Wed Dec 29
last.dump	list	list	4	1043	Wed Dec 29
my.vbar	function	function	7	4056	Wed Dec 29
P8.bd	bdFrame	list	321...	26864	Wed Dec 29

Figure 2.11: *S-PLUS Workbench Objects view*

Outline View

The **Outline View** displays an outline of the elements in the script open in the script editor. In the S-PLUS Workbench, **Outline View** displays functions and objects in the order they appear in the script editor. Items that you have identified to "watch" in the **Functions to watch** text box of the **Preferences** dialog box appear in the **Outline View** with an arrow. You can jump to the definition of a function or object (or other structure element) by clicking it in **Outline View**.

The **Outline View** contains a menu bar that displays the following toggle buttons:

Table 2.3: *Outline View buttons.*

Button	Description
	Click to hide all standard functions displayed in the **Outline View**. Click again to display standard functions.
	Click to hide all functions that you have designated to watch displayed in the **Outline View**.
	Click to hide all anonymous functions displayed in the **Outline View**.
	Click to hide all variables in the **Outline View**.
	Click to sort items displayed the **Outline View** alphabetically. Click again to return the items to the order in which they appear in the script.
	Click to display a menu showing all buttons available on the button bar. (You can toggle these selections either using the menu, or on the button bar.)

.

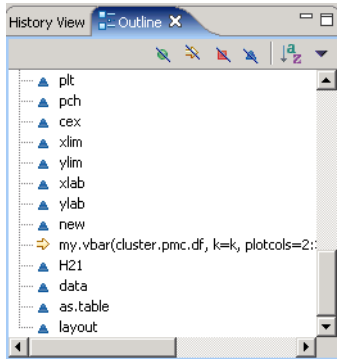


Figure 2.12: *S-PLUS Workbench Outline view*

Output View

The **Output View** displays the code you run (and the results of the code you run) when you click either **Run** on the toolbar, or when you press F9. The text displayed in the **Output View** is replaced each time you click **Run** or press F9. That is, unlike the **Console View**, the **Output View** does not store and display previously-run commands. Also unlike the **Console View**, the **Output View** is not editable; however, you can select and copy lines of text in the **Output View**. You can also print or clear the entire contents of the **Output View**.

You can use the **Output View** control menu (click) to perform the following tasks:

- Clear the contents of the view.
- Copy the selected text.
- Find a string.
- Select all text.
- Save the view contents to a file.

- Print the view contents.

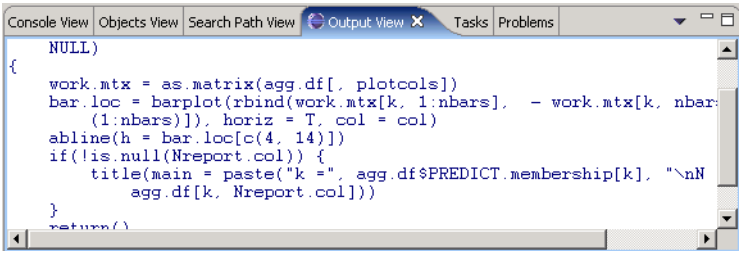


Figure 2.13: *S-PLUS Workbench Output view*

Problems View

The **Problems View** is a standard Eclipse view that displays errors as you edit and save code. For example, if you forget a bracket or a quotation mark, and then save your work, the description appears as a syntax error in the **Problems View**.

Note
Syntax problems appear in the **Problems View** only after you save the file.

If your code has a problem that is displayed in the **Problems View**, and the view is not the active view, the **Problems View** tab title appears as bold text.

To open the Script editor at the location of the problem, double-click the error in the **Problems View**.

You can use the **Problems View** control menu (click ▾) to perform the following tasks:

- Display the **Sorting** dialog box to sort the problems displayed in the view, either in ascending or descending order, and according to the problems' characteristics.

- Display the **Filters** dialog box to specify properties for filtering problems.

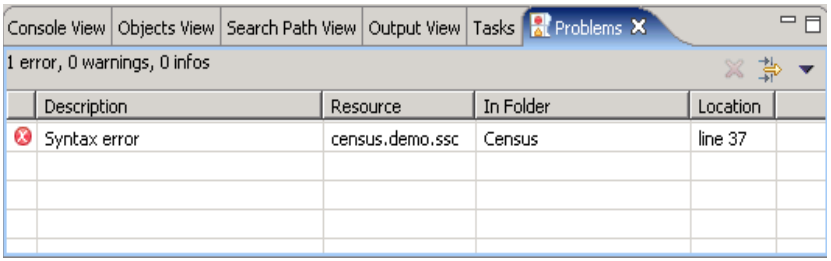


Figure 2.14: *S-PLUS Workbench Problems view*

Search Path View

The **Search Path View** displays the names (or full pathname, in the case of the working data) and search path position of all the attached S-PLUS databases.

By right-clicking the **Search Path View**, you can:

- Attach a library.
- Attach a module.
- Attach a directory.
- Detach the currently-selected database in the view.
- Refresh the current view.

Note
When you use the control menu to add to (or remove from) the **Search Path View** a library, module, or directory, the view automatically refreshes. When you run code to add or remove a library, module, or directory, the view is not automatically refreshed. To refresh the view, right-click the **Search Path View** (or click the control menu button, and then from the menu, click **Refresh**.

The databases that are in your search path determine the objects that are displayed in **Objects View**. That is, if a database is in your search path, the objects in that database appear in the **Objects View**. See the

section Examining Objects on page 51. For more information about working with the **Search Path View**, see the section Changing Attached Databases on page 46.

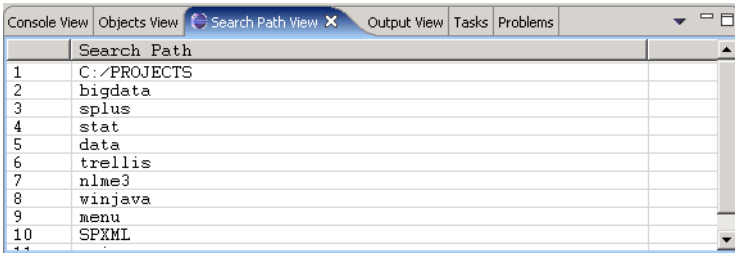


Figure 2.15: *S-PLUS Workbench Search Path view*

Tasks View

The **Tasks View** is a standard Eclipse IDE view, which is customized in S-PLUS to provide three levels of tasks:

Table 2.4: *S-PLUS Workbench Tasks*

Task	Description
FIXME	Defines high-priority tasks. The task appears with an exclamation mark in the **Tasks** view.
TODO	Defines medium-priority tasks.
XXX	Defines low-priority tasks.

You can change these tasks, or you can add your own custom tasks. For more information about changing task settings, see the section To Set the Example Preferences on page 44.

The **Tasks View** also contains a button bar that displays the following buttons:

Table 2.5: *Tasks View buttons.*

Button	Description
	Click to display the **Add Task** dialog box to add a custom task.
	Click to delete the selected custom task. (Note that you cannot use this button to delete tasks identified in the script.)
	Click to display the **Filters** dialog box to specify properties for filtering the tasks.

You can use the **Tasks View** control menu (click) to perform the following tasks:

- Display the **Sorting** dialog box to sort the tasks displayed in the view, either in ascending or descending order, and according to the tasks' characteristics.
- Display the **Filters** dialog box to specify properties for filtering tasks.

For more information about the basic Eclipse **Tasks View**, see the *Workbench User's Guide.*

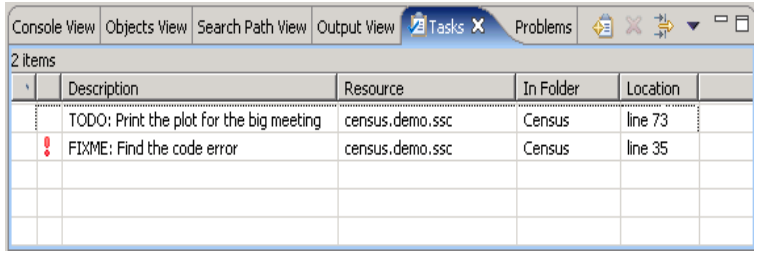


Figure 2.16: *S-PLUS Workbench Tasks view*

SCRIPT EDITOR

The S-PLUS Workbench Script Editor is a text editor. It is similar to the Script Editor in S-PLUS; however, it contains additional script-authoring features such as syntax coloring and integration with the other views in the IDE.

Text Editing Assistance

To help you write efficient, easy-to-follow scripts, the Script Editor provides the following features:

- Displays keywords and function arguments in customizable colors. See the section Setting the Project's Preferences on page 44.
- Displays code line numbers in a column adjacent to the code.
- Provides automatic code indentation and parenthesis matching. (See the Eclipse documentation for more information on the editor's standard features.)
- Provides customized menu items to control text layout and integration in the Script editor.
- Activates the Script **Outline View** when you edit a script.
- Displays the help topic for documented functions when you select the function name, and then type F1.

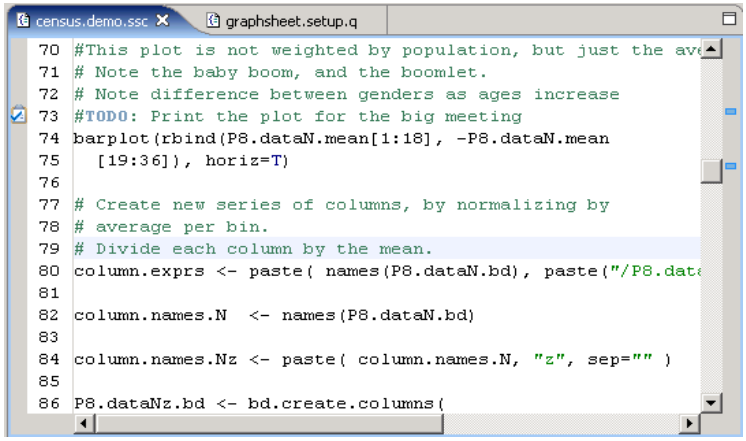

census.demo.ssc
graphsheet.setup.q
70 #This plot is not weighted by population, but just the ave
71 # Note the baby boom, and the boomlet.
72 # Note difference between genders as ages increase
73 #TODO: Print the plot for the big meeting
74 barplot(rbind(P8.dataN.mean[1:18], -P8.dataN.mean
75 [19:36]), horiz=T)
76
77 # Create new series of columns, by normalizing by
78 # average per bin.
79 # Divide each column by the mean.
80 column.exprs <- paste(names(P8.dataN.bd), paste("/P8.data
81
82 column.names.N <- names(P8.dataN.bd)
83
84 column.names.Nz <- paste(column.names.N, "z", sep="")
85
86 P8.dataNz.bd <- bd.create.columns(

Figure 2.17: *S-PLUS Workbench Script editor*

Note

You can use the Eclipse editor to edit non-project files in the S-PLUS Workbench. To open a non-project file, on the **File** menu, click **Open External File**, and then browse to the location of the file to edit. For more information about editing files in Eclipse, see the *Eclipse User's Guide.*

View integration

The Script Editor is closely integrated with the views in the S-PLUS Workbench. This integration includes the following:

- When you type a task keyword in the editor, it is automatically added to the **Tasks View**. See the section Tasks View on page 36 for more information.
- When you make an error and save your script file, the error shows in the **Problems View**. See the section section To Examine Problems on page 54 for more information.
- When you create a new object in the script, it appears in the **Objects View**, with its properties. The object also appears in the **Outline View**.

Menu Options

S-PLUS customizes the basic Eclipse menu and right-click menus to include the following Script Editor control menu items.

Copy to Console

This menu item is available only through the right-click menu. Use this command to copy the text selected in the Script editor to the **Console View**. When you copy text to the **Console View**, S-PLUS runs the command. See the section Copying Script Code to the Console on page 52 for more information.

Run

This menu item is available through the right-click menu. It is also available as a button ( ) on the toolbar, and by pressing F9 when the Script Editor is in focus. Use this command either to run the entire script, or to run the selected commands in the Script editor. When

you run the script, you can observe the results in the **Output View**. See the section Running Code and Reviewing the Output on page 54 for more information.

Note
The S-PLUS Workbench does not implement the core Eclipse **Run** menu item.

S-PLUS Help

This menu item is available from the **Help** menu. When you open S-PLUS Help from the **Help** menu, the S-PLUS Language Reference displays the topic for the `help` function.

Source

The **Source** menu contains the following four submenus:

- **Format**: Applies S-PLUS consistent formatting and line indentation to the entire script.
- **Toggle Comment**: Designates the selected text in the Script editor as a comment, or, if the selected text already is a comment, removes the comment designation.
- **Shift Right**: Moves the selected text four character spaces to the right.
- **Shift Left**: Moves the selected text five character spaces to the left.
- **Open S-Plus Help File**: Opens the S-PLUS Language Reference to the topic for the selected function. If you have no documented function selected, the `help` function topic is displayed.

Source Current File

This menu item is available from the right-click menu in the Script Editor. Selecting this menu option parses and then evaluates each expression in the given file, displaying the results in the **Console View**.

S-PLUS WORKBENCH TASKS

The following topics demonstrate the basic tasks for the S-PLUS Workbench user. For information about basic Eclipse IDE tasks, see the *Eclipse Workbench User's Guide.*

Creating a Project

Before you begin working with files in the S-PLUS Workbench, you must create a project. The S-PLUS Workbench project is a resource containing text files, scripts, and other associated files. You can use the project to control build, version, sharing, and resource management.

Before you create a new project, consider the following scenarios, and then review the S-PLUS Workbench options.

Table 2.6: *S-PLUS Workbench project scenarios.*

Scenario	S-PLUS Workbench Option
You are starting an empty project with no existing files.	In the **New Projec**t wizard, specify a project name and accept the default project directory location. Your project is created as a subdirectory in the Workspace directory. (The **Navigator View** displays the **.project** resource but no existing project files.)
You have one or more project(s), and you want to work with the files at their existing location.	In the **New Project** wizard, specify a project name, clear the **Use default** check box, and then browse to the location of the project files. S-PLUS Workbench works with the files at the specified location. (The **Navigator View** displays the **.project** resource and all files in the project directory.)

Table 2.6: *S-PLUS Workbench project scenarios. (Continued)*

Scenario	S-PLUS Workbench Option
You have an existing project, and you want to copy selected files to a Workspace directory (perhaps, for example, because they are at a remote location, are read-only, or you do not want to work with the original files).	In the **New Project** wizard, specify a project name and accept the default project directory location. An empty project subdirectory is created in the Workspace directory. You can then import your project files. See the section Importing Files on page 43 for more information.

In the following sections, create an empty project, and then import the Census project files.

To Create the Example Project

1. Click **File ▶ New▶ Project**.
2. In the **New Project** dialog box, expand the S-PLUS node and select S-PLUS Project. Click **Next**.
3. Provide the friendly project name, "Census."
4. Accept the option **Use default**. This option creates the project directory in the default Workspace location.
5. Click **Finish** to create the project.

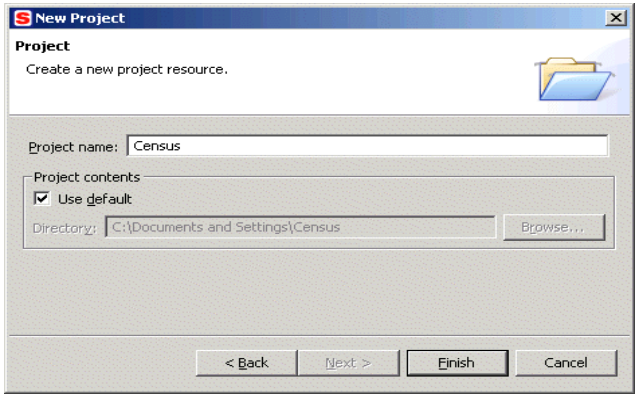


Figure 2.18: *New Project dialog box*

Note

When you create a project, you see in the **Navigator View** the **.project** resource. This resource is created by Eclipse and contains information that Eclipse uses to manage your project. You should not edit this file.

Importing Files

In this exercise, use the Census example, one of the examples provided with the S-PLUS Enterprise Developer edition.

To Import Files

1. With the Census Project node selected in the **Navigator View**, click **File ▶ Import**.
2. In the **Import Select** dialog box, select **File** system, and then click **Next**.
3. In the **Import File system** dialog box, browse to the location of the census project (by default, in your installation directory at **/samples/bigdata/census**.)
4. Select the directory, and then click **OK**. The directory name appears in the left pane, and all of the project's files appear in the right pane.
5. Click **Select All**, and then click **Finish** to add the files to your project.

Hint

You can select just the **.ssc** file to import if you prefer, because the script itself references the data in these files. For the purposes of this part of the exercise, we import all files.

Adding a Second Project

In this exercise, use the Boston Housing example, one of the examples provided with the S-PLUS Enterprise edition. This exercise demonstrates adding a new project at a different location, rather than importing the files.

To Add a Project

1. Click **File ▶ New ▶ Project**.

2. In the **New Project** wizard, select **S-PLUS Project**, and then click **Next**.
3. In the **Project name** text box, type "Boston Housing," and then clear the **Use default** check box.
4. Browse to the location of the Boston Housing sample directory, by default in the **/samples/bigdata** directory of your S-PLUS installation. Select the **boston** directory, and then click **OK**. Click **Finish** to add the project.
5. In the **Navigator View**, the **Boston Housing** directory appears. This directory contains all of the files in that sample directory location.
6. You won't be using this project for the remainder of the tutorial, so right-click the directory, and then select **Delete**.
7. In the **Confirm Delete Project** dialog box, select **Do not delete contents**. (Otherwise, you will delete the sample from your installation directory.)
8. Click **Yes** to remove the project.

Setting the Project's Preferences

S-PLUS provides customizations to the Eclipse IDE to accommodate the specific needs of the S-PLUS programmer.

To Set the Example Preferences

1. On the **Window** menu, click **Preferences**.
2. In the **Preferences** dialog box, expand the Workbench node and examine the dialog box pages.
3. Click **File Associations** and review the file types that the Script Editor recognizes.
4. Click the S-PLUS node.
5. Click **New**.
6. In the **Add New Function to Watch** dialog box, add `set.seed`. Click **OK**.
7. Review the list in the **Functions to Watch** dialog box. Note that `set.seed` has been added to the list.
8. Click **Task Tags**.

9. Highlight the items to change in the **S-PLUS Task Options** text box, or, using the **New**, **Remove**, **Up**, and **Down** buttons, edit the available tasks.
10. Click **OK** or **Apply** to save your changes, or click **Restore Defaults** to return the task options to their default state.
11. Click **OK** to save your changes.

Customizing the S-PLUS Workbench Default Perspective and Views

The default layout of the S-PLUS Workbench presents the **Navigator View**, **Outline View**, and **History View** on the left side of the window. The **Console View**, **Objects View**, **Output View**, **Tasks View**, and **Problems View** are tiled across the bottom of the window. The Script Editor pane is empty.

To Customize the S-PLUS Workbench Default Perspective

1. Click the **Outline View** tab and drag the view beside the **Navigator View**. The **Outline View** now tiles with the **Navigator View**.
2. Click the **History** view tab and drag the view to the right; it now tiles with the other views.
3. Right-click the **Tasks** view tab and select **Fast View**. The **Tasks** view minimizes and appears as an icon in the window's status bar.
4. Click the **Console** view tab to select it.
5. Click **Window ▶ Save Perspective As**.
6. In the **Name** box, type "Sample Exercise," and then click **OK**.

The **Sample Exercise** perspective button appears on the toolbar:

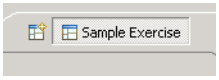


Figure 2.19: *Sample exercise perspective button*

To return to the S-PLUS Workbench default, click the perspective button to the left of the **Sample Exercise** button, and then click **Other**.

In the **Select Perspective** dialog box, select S-PLUS (default), and then click **OK**. The perspective returns to its previous layout.

You can select other views to display in your perspective.

To Change the Displayed Views

1. To change the views, or to display the list of available views, on the menu, click **Window ▶ Show View**.
2. From the submenu, select the view to display.

   Alternatively, if you do not see the view you want to display, from the **Show View** menu, click **Other**, and then select a view from the **Show View** dialog box.

   - If the view is not currently visible in the UI, selecting it displays the view and gives it focus in the UI.
   - If the view is available, selecting it gives it focus in the UI.

Changing Attached Databases

S-PLUS recognizes libraries, modules, and directories as legitimate object databases. You can add and detach any of these types of databases to the **Search Path View**.

By default, the **Search Path View** displays the full path of the working database and all of the attached S-PLUS data libraries. Objects existing in a recognized active database appear in the **Objects View**.

Adding a Database

Objects in an added database appear in **Objects View** when you refresh the view to that database. See the section Examining Objects on page 51.

To Add a Library

1. Right-click the **Search Path View**.
2. From the right-click menu, click **Add Library**.
3. In the **Attach Library** dialog box, type MASS. Clear the **Attach at top of search list** check box to indicate that you want add the library to the bottom position.
4. Click **OK** and examine the **Search Path View** for the change.

To Add a Module

1. From the right-click **Search Path** menu, click **Add Module**.
2. In the **Attach Module** dialog box, provide the module name and indicate whether to add it to the first position.
3. Click **OK** and examine the **Search Path View** for the change.

To Add a Directory

1. Right-click the **Search Path View**.
2. From the menu, click **Attach Directory**.
3. In the **Attach** dialog box, in the **Directory to attach** text box, browse to the directory location.
4. In the **Label** text box, type `Projects`
5. In the **Position** text box, type `4`.
6. Click **OK** and examine the **Search Path View**. The label you provided should appear at position 4.

Detaching a Database

From the **Search Path View**, you can detach a database from your current session.

To Detach a Database

1. In the **Search Path View**, right-click **bigdata**.
2. In the right-click menu, select **Detach**.
3. Examine the **Search Path View**. The Big Data library is no longer attached.

Refreshing the View

When you refresh the view, any changes to the **Search Path View** that have not been reflected in a recent change are displayed. For example, if you add a library by calling the load function in an S-PLUS script, the change is not immediately displayed in the **Search Path View**.

To Refresh the View

1. Using the **Console View**, reattach the Big Data library. In the **Console View**, type

```
library(bigdata, first = T)
```

2. Right-click the **Search Path View**.
3. In the right-click menu, click **Refresh**. Notice that the Big Data library appears as attached in the first position (position 2).

Creating a Script

You can create a new S-PLUS script file, or you can import an existing script file. The following two examples demonstrate both techniques.

To Create a New Script File

1. Click **File ▶ New ▶ Other**.
2. In the **New** dialog box, expand the Simple node and select File. Click **Next**.
3. In the **New File** dialog box, select the parent directory (the Stock Project directory)
4. In the **File name** text box, type **Sample.ssc**.
5. Click **Finish** to create the file.

We won't work with this file for this exercise, so you can either disregard the file, or delete it from your project. Alternatively, you can open the file, add some S-PLUS code, and save it in the project.

Viewing Project Files

The **Navigator View** displays the project files. In Windows, if you have Microsoft Excel installed, you can open a CSV file in an external window. In this project, only the files identified in **Windows ▶ Preferences** in the File Extensions page open in the Script editor.

Removing files from a project

Because the project script imports the data in the files from their installation directory in S-PLUS, you don't need to have them all in the project. However, removing an imported file deletes it from your project directory, so remove individual files with care.

To Remove a File

1. In the **Navigator View**, select all files except **census.demo.ssc**.
2. Right-click the selected files, and then click **Delete**.

3. In the **Confirm Resource Delete** dialog box, click **OK** to remove the files from the project. The **Navigator View** should now just display the Census Project directory, the project file, and **census.demo.ssc**:

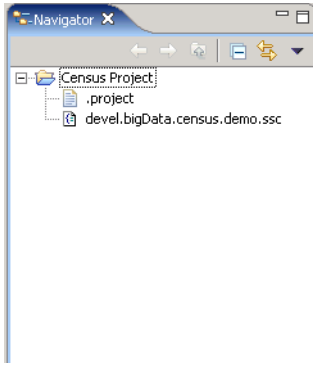


Figure 2.20: *Navigator view after deleting files*

Editing Code in the Script Editor

The S-PLUS script is a text file that you can edit in the Script Editor. In this exercise, just edit **census.demo.ssc** using the menu items provided specifically for S-PLUS.

To Edit Script Code

1. In the **Navigator View**, double-click the file **census.demo.ssc** to open it in the Script Editor and examine the script. Note that:
   - The comment text appears in the Script Editor as green. (You can change this default color in the **Preferences** dialog box. See the Eclipse User's Guide and the section Setting the Project's Preferences on page 44 for more information.)
   - The line that has focus appears highlighted.
   - The line numbers appear to the left of the script text.
2. Scroll to line 12 and highlight the line and the next line:

```
stringsAsFactors=F,
startRow=1, bigdata=T).
```

3. Click **Source ▶ Shift Left**. The code shifts four character spaces to the left.

4. Click **Source ▶ Format**. This command formats the entire script. Note that the formatting change you made in the previous step has been reverted. Also note that the line numbers for formatted functions are highlighted.

Hint
The line numbers for any line changed in your script are highlighted until the next time you save your work.

5. Scroll to line containing the comment

   ```
   #bd.data.viewer(P8.bd)
   ```

6. Click **Source ▶ Toggle Comment** to remove the comment character. (Alternatively, you can just delete the comment character.)
7. Notice that the script text color changes to indicate that the line is no longer a comment.
8. Scroll to line 187. Select all rows from 187 through the end of the script, and then click **Source ▶ Toggle Comment**. (The graphsheet will not launch from the S-PLUS Workbench.)

Examining the Outline

The **Outline View** displays all of the items (objects, functions, and so on) that are contained in the open script. **Outline View** is not editable.

To Examine the Outline

1. Examine the objects that appear in the **Outline View**. Note that `set.seed` appears with a yellow arrow next to it, because in the section Setting the Project's Preferences on page 44, you indicated that `set.seed` was a function to watch.
2. Scroll through the **Outline View** list and highlight an object. Note that the Script Editor scrolls to, and highlights, the line where the object appears.

Examining Objects

Details about your project's objects (and all objects in your database) appear in the **Objects View**. **Objects View** is not editable; however, you can refresh the contents or change the view to another attached database. To refresh the view, right-click the **Objects View** and click **Refresh**.

To Examine the Objects

1. Select the **Objects View** tab to display the objects and their details. By default, the objects are displayed sorted by name.
2. Right-click the **Objects View** and, in the right-click menu, click **bigdata**. The Big Data library objects are displayed in the **Objects View**. (It might take a few seconds to display all of the objects.)
3. Resort the objects by any property displayed in the **Objects View** by clicking the property's column title.

To Select Another Object Database

1. Right-click the **Objects View** and, in the right-click menu, click your default object directory (the first database in the list, by default found in your installation directory at **/users/*yourname***). The project objects are displayed in the **Objects View**. (It might take a few seconds to display all of the objects.)

Adding a Task to A Script

The **Tasks View** displays outstanding project tasks As discussed in the section Setting the Project's Preferences on page 44, the indicators for task levels are stored in the **Preferences** dialog box. (Click **Windows ▶ Preferences** to display them.) You can add a task in one of two ways:

- Add the task directly to the **Tasks View**.
- Add the task to the script file.

To Add a Task Directly to the Tasks View

1. Click the **Tasks View** tab to give it focus.
2. Right-click the view, and then click **Add Task**.
3. In the **Add Task** dialog box, provide the description and priority level of the task.

4. Click **OK** to save and display the new task.

A task added directly to the **Tasks View** displays a check box (for marking the task complete) in the **Tasks View**'s first column. It does not display a reference to a resource, a directory, or a location.

To Add a Task in the Script File

In the script file, scroll to line 6.

1. Type the following text:

   ```
   #FIXME: Remove the comment markers to display the
   viewer
   ```

2. Save the script file.

   Note that the FIXME comment appears in the **Tasks View** as a high-level task, with a red exclamation mark in its second column. The task also displays information about its resource, directory, and line location. You can go directly to any task in your script by double-clicking it in the **Tasks View**.

3. In the Script Editor, change the level of the task by changing FIXME to TODO and save the file. Note that the exclamation mark disappears, and the task becomes a normal level task.

Running Code

You can run your S-PLUS script code directly from Eclipse in two ways:

- Copy a selected block of code from the Script Editor to the **Console View**.
- Run the selected code (or all code, if none is selected) by clicking **Run** or pressing F9.

Copying Script Code to the Console

The **Console View** is an editable view (in other words, you can type commands and run them by pressing ENTER); therefore, when you copy script contents to the **Console View**, you must include the line return, or the script will not run. This behavior is consistent with the S-PLUS **Commands** window, in the S-PLUS GUI, which also requires a line return to run code.

Also like the S-PLUS **Commands** window, the **Console View** concatenates the code that runs throughout your S-PLUS Workbench session, so you can review and save it.

To Run Copied Script Code

1. Select lines 1 and 9 in the script. Be sure to select the line return at the end of line 9.
2. Right-click the code and click **Copy to Console**. The selected code is copied immediately to the **Console View** and runs. You do not need to paste it in the **Console View**.
3. Repeat steps 1 and 2 for line 10.
4. Finally, repeat steps 1 and 2 for lines 11-13.

   (You can select all of the code, lines 1-13, but if you do so, it appears in the **History View** as one line. By following the steps above, the **History View** reflects the three different calls to run the code. See the section Examining the History View on page 53 for more information.)

Examining the History View

This exercise uses the script code run in the section Copying Script Code to the Console on page 52.

The **History View** reflects the code run in the **Console View**. Note that the **History View** displays each selection you make, even if it is more than one command, on one line, and if the line extends beyond about 50 characters, the **History View** displays an ellipse (...) to indicate more code. To display each line of code in the **History View**, you must run the lines individually.

To Examine the History

1. To examine and rerun code from the **History View**.
2. Click the **History View** tab to give it focus.
3. Right-click the first line of code, and click **Select input**. The code is copied to the **Console View**. You must return to the **Console View** and press ENTER to run the code.

   (Alternatively, double-click the code in the **History View** to copy it to the **Console View**.)

You can scroll through the individual entries in the **History View**; as you scroll, the selection appears in the **Console View**. To run a selected item, switch from the **History View** to the **Console View** and press ENTER at the end of the code line.

Running Code and Reviewing the Output

You can run code directly from the Script Editor by using the **Run** feature.

To Run Code

1. Select the **Output View** tab.
2. In the Script Editor, select the code to run (or, to run the whole script, select nothing), and press F9, or on the toolbar, click **Run**.

   The **Output View** displays the run code and any S-PLUS messages.

Fixing Problems in the Code

Introduce a programmatic problem in the script to examine the results in the **Problems View**.

To Examine Problems

1. In the Script Editor, on line 9 of the script, remove the closing parenthesis.
2. Save the file. Note that the **Problems View** tab shows bold text.
3. Click the **Problems View** tab to display the view.
4. Click the problem description. Note that the Script Editor highlights the line where the code is broken.
5. In the Script Editor, replace the missing parenthesis and save your file. Note that the problem disappears from the **Problems View**.

Closing the Project

The S-PLUS Workbench maintains a list of your active projects in the **Navigator View**, even after you close all associated files.

To Close the Project

1. Click **File ▶ Close All**
2. Examine the views and note that the views all still contain data.

   The views continue to show project information. The S-PLUS Workbench stores information in many views, even after you close the interface. For example, the **Objects View** continues

to store information about all your projects' objects, and the **Tasks View** and **Problems View** continue to display outstanding issues. These features can help you track outstanding work, even between sessions.

COMMONLY-USED FEATURES IN ECLIPSE

The core Eclipse IDE contains many additional features that you might find helpful in managing your projects. The following table lists a few of these features, along with references to the *Eclipse Workbench User Guide* to help you learn how to use them effectively.

Table 2.7: *Eclipse Tasks and Features.*

Task	Eclipse Feature Description
Comparing files with previous versions.	The **Compare With Local History** menu item is available from the control menu in **Navigator View**. Using this feature, you can compare the current version of the selected file with previously-stored local versions. For more information, see the topic "Local history" in the *Eclipse Workbench User Guide.*
Replacing files with a previous version.	The **Replace With Local History** and **Replace With Previous from Local History** menu items are available from the control menu in **Navigator View**. Using these features, you can replace the current version of the selected file with one of the previously-stored local versions. **Replace With Previous from Local History** displays no selection dialog box; it just replaces the file. To choose a previous state in the **Local History** list, use **Replace With Local History**. For more information, see the topic "Replacing a resource with local history" in the *Eclipse Workbench User Guide.*

Table 2.7: *Eclipse Tasks and Features. (Continued)*

Task	Eclipse Feature Description
Finding a word in a project or a term in a Help topic.	Using the **Search ▶ File** menu item, you can find all occurrences of a word in a project or Help topic. For more information, see the topic "File search" in the *Eclipse Workbench User Guide.*
Filter files in the **Navigator View**.	Using the **Working Sets** menu option on the control menu in **Navigator View**, you can create subsets of files to display or hide. For more information, see the topics "Working Sets" and "Showing or hiding files in the Navigator View" in the *Eclipse Workbench User Guide.*
View a file that is not part of your project.	Use the **File ▶ Open External File** menu item to open a file that is not part of your project.

THE BIG DATA LIBRARY 3

Introduction **60**

Working with a Large Data Set **61**
Finding a Solution 61
No 64-Bit Solution 64

Size Considerations **65**
When to Set `bigdata=T` 65
Summary 68

The Big Data Library Architecture **69**
Block-based Computations 69
Data Types 73
Classes 77
Functions 78
Summary 83

INTRODUCTION

In this chapter, we discuss the history of the S language and large data sets and describe improvements that the Big Data library presents. This chapter discusses data set size considerations, including when to use the Big Data library. The chapter also describes in further detail the Big Data library architecture: its data objects, classes, functions, and advanced operations.

WORKING WITH A LARGE DATA SET

When it was first developed, the S programming language was designed to hold and manipulate data in memory. Historically, this design made sense; it provided faster and more efficient calculations and modeling by not requiring the user's program to access information stored on the hard drive. Data size has outstripped the rate at which RAM size increased; consequently, S program users could have encountered an error similar to the following:

```
   Problem in read.table: Unable to obtain requested dynamic
memory.
```

This error occurs because S-PLUS requires the operating system to provide a block of memory large enough to contain the contents of the data file, and the operating system responds that not enough memory is available.

While S-PLUS can access data contained in virtual memory, the maximum size of data files depends on the amount of virtual memory available to S-PLUS, which depends in turn on the user's hardware and operating system. In typical environments, virtual memory limits your data file size, and then it returns an out-of-memory error.

Finally, you can also encounter an out-of-memory error after successfully reading in a large data object, because many S functions require one or more temporary copies of the source data in RAM for certain manipulation or analysis functions.

Finding a Solution

S programmers with large data sets have historically dealt with memory limitations in a variety of ways. Some opted to use other applications, and some divided their data into "digestible" batches, and then recompile the results. For S programmers who like the flexibility and elegant syntax of the S language and the support provided to owners of an S-PLUS license, the option to analyze and model large data sets in S has been a long-awaited enhancement.

Out-of-Memory Processing

The Big Data library, available in S-PLUS Enterprise Developer, provides this enhancement by processing large data sets using scalable algorithms and data streaming. Instead of loading the contents of a large data file into memory, S-PLUS creates a special

binary cache file of the data on the user's hard disk, and then refers to the cache file on disk. This out-of-memory design requires relatively small amounts of RAM, regardless of the total size of the data.

Scalable Algorithms

Although the large data set is stored on the hard drive, the scalable algorithms of the Big Data library are designed to optimize access to the data, reading from disk a minimum number of times. Many techniques require a single pass through the data, and the data is read from the disk in blocks, not randomly, to minimize disk access times. These scalable algorithms are described in more detail in the section The Big Data Library Architecture on page 69.

Data Streaming

S-PLUS operates on the data binary cache file directly, using "streaming" techniques, where data flows through the application rather than being processed all at once in memory. The cache file is processed on a row-by-row basis, meaning that only a small part of the data is stored in RAM at any one time. It is this out-of-memory data processing technique that enables S-PLUS to process data sets hundreds of megabytes, or even gigabytes, in size without requiring large quantities of RAM.

New Data Type

S-PLUS Enterprise Developer introduces the large data frame, an object of class `bdFrame`. A big data frame object is similar in function to standard S-PLUS data frames, except its data is stored in a cache file on disk, rather than in RAM. The `bdFrame` object is essentially a reference to that external file: While you can create a `bdFrame` object that represents an extremely large data set, the `bdFrame` object itself requires very little RAM.

For more information on `bdFrame`, see the section Data Frames on page 73.

S-PLUS Enterprise Developer also introduces time date (`bdTimeDate`), time span (`bdTimeSpan`), and series (`bdSeries`, `bdSignalSeries`, and `bdTimeSeries`) support for large data sets. For more information, see the section Time Date Creation on page 235, in the Appendix.

Flexibility

The Big Data library provides reading, manipulating, and analyzing capability for large data sets using the familiar S programming language. Because most existing data frame methods work in the same way with `bdFrame` objects as they do with `data.frame` objects, the style of programming is familiar to S-PLUS programmers. Much existing code from previous versions of S-PLUS runs without

modification in the Big Data library, and only minor modifications are needed to take advantage of the big-data capabilities of the pipeline engine.

Balancing Scalability with Performance

While accessing data on disk (rather than in RAM) allows for scalable statistical computing, some compromises are inevitable. The most obvious of these is computation speed. The Big Data library in the S-PLUS Enterprise Developer provides scalable algorithms that are designed to minimize disk access, and therefore provide optimal performance with out-of-memory data sets. This makes S-PLUS Enterprise Developer a reliable workhorse for processing very large amounts of data. When your data is small enough for traditional S-PLUS, it's best to remember that in-memory processes are faster than out-of-memory processes.

If your data set size is not extremely large, all of the S-PLUS traditional in-memory algorithms remain available, so you need not compromise speed and flexibility for scalability when it's not needed.

Metadata

To optimize performance, S-PLUS stores certain calculated statistics as metadata with each column of a `bdFrame` object and updates the metadata every time the data changes. These statistics include the following:

- Column mean (for numeric columns).
- Column maximum and minimum (for numeric and date columns).
- Number of missing values in the column.
- Frequency counts for each level in a categorical column.

Requesting the value of any of these statistics (or a value derived from them) is essentially a free operation on a `bdFrame` object. Instead of processing the data set, S-PLUS just returns the precomputed statistic. As a result, calculations on columns of `bdFrame` objects such as the following examples are practically instantaneous, regardless of the data set size. For example:

- `mean(census.data$Income)`
- `range(census.data$Age)`

No 64-Bit Solution

Are out-of-memory data analysis techniques still necessary in the 64-bit age? While S-PLUS Enterprise Developer is available on some 64-bit systems, the out-of-memory techniques described above are still required to analyze truly large data sets.

64-bit systems increase the amount of memory that the system can address. This can help in-memory algorithms handle larger problems, provided that all of the data can be in physical memory. If the data and the algorithm require virtual memory, page-swapping (that is, accessing the data in virtual memory on the disk) can have a severe impact on performance.

With data sets now in the multiple gigabyte range, out-of-memory techniques are essential. Even on 64-bit systems, out-of-memory techniques can dramatically outperform in-memory techniques when the data set exceeds the available physical RAM.

SIZE CONSIDERATIONS

While the Big Data library imposes no predetermined limit for the number of rows allowed in a big data object or the number of elements in a big data vector, your computer's hard drive must contain enough space to hold the data set and create the data cache. Given sufficient disk space, the big data object can be created and processed by any scalable function.

The speed of most Big Data library operations is proportional to the number of rows in the data set: if the number of rows doubles, then the processing time also doubles.

The amount of RAM in a machine imposes a predetermined limit on the number of columns allowed in a big data object, because column information is stored in the data set's metadata. This limit is in the tens of thousands of columns. If you have a data set with a large number of columns, remember that some operations (especially statistical modeling functions) increase at a greater than linear rate as the number of columns increases. Doubling the number of columns can have a much greater effect than doubling the processing time. This is important to remember if processing time is an issue.

Note

When you import data, you have the option to set the flag `stringsAsFactors` to `T` or `F` (the default is `T`). S-PLUS imposes a limit of 500 levels for `bdFactors`.

When to Set `bigdata=T`

When you get ready to import data into an S-PLUS session, you might find yourself considering whether to use the Big Data library, or to use the standard S-PLUS library. Using the standard S-PLUS library can provide you with faster processing times, because working in-memory is more efficient than streaming the data when virtual memory is not in use. However, if your data is large, when you try to import the data, the process is very slow because of necessary swapping, or worse, you run the risk of trying to import the data and seeing the error message:

```
Unable to Obtain Requested Dynamic Memory
```

For standard S-PLUS, the absolute upper limit on the size of datasets it can work with is set by the maximum amount of memory that S-PLUS can address. On 32-bit systems, this theoretical limit is 2^32 bytes, or approximately 4 GB. There is a practical limit that is determined by the operating system (that is, the operating system requires some of the aforementioned 4 GB). For example, a 32-bit Windows system without special configuration reduces available virtual memory to about 1.5 GB.

In addition to considering the initial size of the data set, you must also consider the numbers of copies that S-PLUS makes while processing the data. The underlying S Language that is part of S-PLUS makes between four and five temporary copies of a dataset in memory.

Memory Requirements for In-Memory Calculations

Memory requirements depend on the following:

- The size of data, including the number of rows and columns in the raw data file.
- Column types; that is, numeric data requires 8 bytes per value, while character data consisting of long strings requires more than 8 bytes per value.
- The data operations to be performed. During data operations, the data needs to be copied on average 4.5 times.

To determine approximately how much total memory (physical and virtual) a dataset requires, use the following formula:

$$r * c * 8 * 4.5 = \text{number of bytes required for the data}$$

where:

- r = number of rows in the input file
- c = number of columns in the input file
- 8 = bytes per entry required for numeric data
- 4.5 = average number of data copies that the S language creates while processing the data.

This formula can give you an idea of the amount of dynamic memory needed in standard S-PLUS. For example, using this formula, you can see that a dataset with 98672 rows and 507 columns of numeric data requires about 1.8 GB of RAM in the processing machine:

*98672 * 507 * 8 * 4.5 = 1,800,961,344 bytes, or approximately 1.8 GB*

On a windows machine, 1.8 GB is approaching the limits of the 32-bit operating system, so you should set `bigdata=T` when importing this data set.

Physical RAM vs. Virtual RAM

For efficient operations, it is best to have space for all of your data in physical memory. If your data requires 1.2 GB of memory (according to the above formula), and you have only 512 MB of RAM and 2 GB of swap space (virtual memory), then performance will likely suffer. For example, a Windows machine in this situation will often appear to hang.

Whenever your memory requirement is more than available physical RAM, you can benefit from moving to out-of-memory processing techniques, such as using the Big Data library.

For more information about how S-PLUS allocates memory, and how to use it effectively, See Chapter 16, Using Less Time and Memory in the *Application Developer's Guide*.

Summary

By bringing together flexible programming and big-data capability, the S-PLUS Enterprise Developer is a data analysis environment that provides both rapid prototyping of analytic applications and a scalable production engine capable of handling datasets hundreds of megabytes, or even gigabytes, in size.

In the next section, we provide an overview to the Big Data library architecture, including data types, functions, and naming conventions.

THE BIG DATA LIBRARY ARCHITECTURE

The Big Data library is a separate library from the S-PLUS engine library. It is designed so that you can work with large data objects the same way you work with existing S-PLUS objects, such as data frames and vectors. The library uses terminology familiar to S-PLUS users and follows these conventions:

- Class names are mixed-case delimited and prepended with the designation `bd`, such as `bdFrame`.
- Function names are period delimited, except when they match an existing function, such as `importData`, or if they refer to a class, such as `as.bdVector`.
- Functions start with `bd.`, such as `bd.compare`, unless the function uses the same syntax as functions available for non-big-data functions, such as the `predict` or `summary` function.
- Big Data library functions do not restrict the number of rows in the data. Because summary information (metadata) is computed and stored for each column, the number of columns is slightly limited. The current implementation supports tens of thousands of columns on a typical computer.
- Function names starting with `bd.internal` are not intended to be used directly.

Block-based Computations

Data sets that are much larger than the system memory are manipulated by processing one "block" of data at a time. That is, if the data is too large to fit in RAM, then the data will be broken into multiple data sets and the function will be applied to each of the data sets. As an example, a 1,000,000 row by 10 column data set of double values is 76MB in size, so it could be handled as a single data set on a machine with 256MB RAM. If the data set was 10,000,000 rows by 100 columns, it would be 7.4GB in size and would have to be handled as multiple blocks.

Table 3.1 lists a few of the optional arguments for the function `bd.options` that you can use to set limits for caching and for warnings:

Table 3.1: *bd.options block-based computation arguments.*

bd.option argument	**Description**
`block.size`	The block size (in number of rows), the number of bytes in the cache to be converted to a `data.frame`.
`max.convert.bytes`	The maximum size (in bytes) of the big data cache that can be converted to a `data.frame`.
`convert.warn`	If `T`, generates a warning whenever a big data cache is converted to a `data.frame`.
`max.block.mb`	The maximum number of megabytes used for block-processing buffers. If the specified block size requires too much space, the number of rows is reduced so that the entire buffer is smaller than this size. This prevents unexpected out-of-memory errors when processing wide data with many columns. The default value is 10.

The function `bd.options` contains other optional arguments for controlling column string width, display parameters, factor level limits, and overflow warnings. See its help topic for more information.

The Big Data library also contains functions that you can use to control block-based computations. These include the functions in Table 3.2. For more information and examples showing how to use these functions, see their help topics.

Table 3.2: *Block-based computation functions.*

Function name	Description
`bd.aggregate`	Use `bd.aggregate` to divide a data object into blocks according to the values of one or more of its columns, and then apply aggregation functions to columns within each block. `bd.aggregate` takes two required arguments: `data`, which is the input data set, and `by.columns`, which identifies the names or numbers of columns defining how the input data is divided into blocks. Optional arguments include `columns`, which identifies the names or numbers of columns to be summarized, and `methods`, which is a vector of summary methods to be calculated for columns. See the help topic for bd.aggregate for a list of the summary methods you can specify for `methods`.
`bd.block.apply`	Run an S-PLUS script on blocks of data, with options for reading multiple input datasets and generating multiple output data sets, and processing blocks in different orders. See the help topic for `bd.block.apply` for a discussion on processing multiple data blocks.
`bd.by.group`	Apply the specified S-PLUS function to multiple data blocks within the input dataset.

Table 3.2: *Block-based computation functions. (Continued)*

Function name	Description
`bd.by.window`	Apply the specified S-PLUS function to multiple data blocks defined by a moving window over the input dataset. Each data block is converted to a `data.frame`, and passed to the specified function. If one of the data blocks is too large to fit in memory, an error occurs.
`bd.split.by.group`	Divide a dataset into multiple data blocks, and return a list of these data blocks
`bd.split.by.window`	Divide a dataset into multiple data blocks defined by a moving window over the dataset, and return a list of these data blocks.

For a detailed discussion on advanced topics, such as block size issues and increasing efficiency, see Chapter 7, Advanced Programming Information.

Data Types

S-PLUS Enterprise Developer introduces the following new data types, described in more detail below:

Table 3.3: *New data types and data names for S-PLUS.*

Big Data class	Data type
`bdFrame`	Data frame
`bdVector`, `bdCharacter`, `bdFactor`, `bdLogical`, `bdNumeric`, `bdTimeDate`, `bdTimeSpan`	Vector
`bdLM`, `bdGLM`, `bdPrincomp`, `bdCluster`	Models
`bdSeries`, `bdTimeSeries`, `bdSignalSeries`	Series

Data Frames

The main object to contain your large data set is the big data frame, an object of class `bdFrame`. Most methods commonly used for a `data.frame` are also available for a `bdFrame`. Big data frame objects are similar to standard S-PLUS data frames, except in the following ways:

- A `bdFrame` object stores its data on disk, while a `data.frame` object stores its data in RAM. As a result, a `bdFrame` object has a much smaller memory footprint than a `data.frame` object.
- A `bdFrame` object does not have row labels, as a `data.frame` object does. While this means that you cannot refer to the rows of a `bdFrame` object using character row labels, this design reduces storage requirements and improves performance by eliminating the need to maintain unique row labels.
- A `bdFrame` object can contain columns of only types `double`, `character`, `factor`, `timeDate`, `timeSpan` or `logical`. No other column types (such as matrix objects or user-defined classes) are allowed. By limiting the allowed column types, S-PLUS ensures that the binary cache file representing the data is as compact as possible and can be efficiently accessed.
- If you use the `$` operator to refer to a column name that is not a syntactic name in S, you must surround it in quotes. For example, `my.bdFrame$"Return(percent)"`.

- The `print` function works differently on a `bdFrame` object than it does for a data frame. It displays only the first few rows and columns of data instead of the entire data set. This design prevents accidentally generating thousands of pages of output when you display a `bdFrame` object at the command line.

Note

You can specify the numbers of rows and columns to print using the `bd.options` function. See `bd.options` in the S-PLUS Language Reference for more information.

- The `summary` function works differently on a `bdFrame` object than it does for a data frame. It calculates an abbreviated set of summary statistics for numeric columns. This design is for efficiency reasons: `summary` displays only statistics that are precalculated for each column in the big data object, making `summary` an extremely fast function, even when called on a very large data set.
- Some data frame methods are not defined for `bdFrame` objects. To use these methods, you must convert your data to a regular data frame. To learn how to convert your data, see the section Converting Data on page 95 of Chapter 4, Exploring and Manipulating Large Data Sets.

Vectors

The S-PLUS Big Data library also introduces `bdVector` and six subclasses, which represent new vector types to support very long vectors. Like a `bdFrame` object, the big vector object stores data out-of-memory as a cache file on disk, so you can create very long big vector objects without needing a lot of RAM.

You can extract an individual column from a `bdFrame` object (using the `$` operator) to create a large vector object. Alternatively, you can generate a large vector using the functions listed in Table A.3 in the Appendix. Like `bdFrame` objects, the actual data is stored out of memory as a cache file on disk, so you can create very long big vector

objects without worrying about fitting them into RAM.The Big Data library vector data types are listed in Table 3.4, along with their corresponding S-PLUS types:

Table 3.4: *bdVector data types.*

Big Data library vector data types	**Analogous classes in S-PLUS**
bdCharacter	character
bdNumeric	double
bdFactor	factor
bdLogical	logical
bdTimeDate	timeDate
bdTimeSpan	timeSpan

You can use standard vector operations, such as selections and mathematical operations, on these data types. For example, you can create new columns in your data set, as follows:

```
census.data$adjusted.income <- log(census.data$income -
  census.data$tax)
```

Models

S-PLUS Enterprise Developer Big Data library introduces scalable modeling algorithms to process big data objects using out-of-memory techniques. With these modeling algorithms, you can create and evaluate statistical models on very large data sets.

The low-level modeling functions in the big data library return a big data model object. This object contains a reference to the bdFrame used to fit the model and a reference to a description of the model.

A model object is available for each of the following statistical analysis model types.

Table 3.5: *Big Data library model objects.*

Model Type	Model Object
Linear regression	`bdLm`
Generalized linear models	`bdGlm`
Clustering	`bdCluster`
Principal Components Analysis	`bdPrincomp`

When you perform statistical analysis on a large data set with the Big Data library, you can use familiar S-PLUS modeling functions and syntax, but you supply a `bdFrame` object as the data argument, instead of a data frame. This forces out-of-memory algorithms to be used, rather than the traditional in-memory algorithms.

When you apply the modeling function `lm` to a `bdFrame` object, it produces a model object of class `bdLm`. You can apply the standard `predict`, `summary`, `plot`, `residuals`, `coef`, `formula`, `anova`, and `fitted` methods to these new model objects.

For more information on statistical modeling, see Chapter 6, Modeling Large Data Sets.

Series Objects

The standard S-PLUS library contains a series object, with two subclasses: `timeSeries` and `signalSeries`. The series object contain:

- A `data` component that is typically a data frame.
- A `positions` component that is a `timeDate` or `timeSequence` object (`timeSeries`), or a `bdNumeric` or `numericSeries` object (`signalSeries`).
- A `units` component that is a character vector with information on the units used in the data columns.

The Big Data library equivalent is a `bdSeries` object with two subclasses: `bdTimeSeries` and `bdSignalSeries`. They contain:

- A `data` component that is a `bdFrame`.
- A `positions` component that is a `bdTimeDate` object (`bdTimeSeries`), or `bdNumeric` object (`bdSignalSeries`).
- A `units` component that is a character vector.

For more information about using large time series objects and their classes, see the section Time Classes on page 81 and the section Working with Time Series Data on page 109 of Chapter 4, Exploring and Manipulating Large Data Sets.

Classes

The Big Data library follows the same object-oriented design as the standard S-PLUS Sv4 design. For a review of object-oriented programming concepts, see Chapter 8, Object-Oriented Programming in S-Plus in the *Programmer's Guide*.

Each object has a class that defines methods that act on the object. The library is extensible; you can add your own objects and classes, and you can write your own methods.

The following classes are defined in the Big Data library. For more information about each of these classes, see their individual help topics.

Table 3.6: *Big Data classes.*

Class(es)	**Description**
`bdFrame`	Big data frame
`bdLm`, `bdGlm`, `bdCluster`, `bdPrincomp`	Rich model objects
`bdVector`	Big data vector
`bdCharacter`, `bdFactor`, `bdLogical`, `bdNumeric`, `bdTimeDate`, `bdTimeSpan`	Vector type subclasses
`bdTimeSeries`, `bdSignalSeries`	Series objects

Functions

In addition to the standard S-PLUS functions that are available to call on large data sets, the Big Data library includes functions specific to big data objects. These functions include the following.

- Big vector generating functions
- Data exploration and manipulation functions.
- Traditional and Trellis graphics functions.
- Modeling functions.

The functions for these general tasks are listed in the Appendix.

Data Import and Export

Two of the most frequent tasks using S-PLUS are importing and exporting data. The functions are described in Table A.1 of the Appendix. You can perform these tasks either from the **Commands** window or from the S-PLUS import and export dialog boxes. For more information about importing and exporting large data sets, see the section Importing Existing Data and the section Exporting Data in Chapter 4, Exploring and Manipulating Large Data Sets.

Big Vector Generation

To generate a vector for a large data set, call one of the S-PLUS functions described in Table A.3 in the Appendix. When you set the `bigdata` flag to `TRUE`, the standard S-PLUS functions generate a `bdVector` object of the specified type. For example:

```
# sample of size 2000000 with mean 10*0.5 = 5
rbinom(2000000, 10, 0.5, T)
```

Data Exploration Functions

After you import your data into S-PLUS and create the appropriate objects, you can use the functions described in Table A.4 in the Appendix to compare, correlate, crosstabulate, and examine univariate computations.

Data Manipulation Functions

After you import and examine your data in S-PLUS, you can use the data manipulation functions to append, filter, and clean the data. For an overview of these functions, see Table A.5 in the Appendix. For a more in-depth discussion of these functions, see Chapter 4, Exploring and Manipulating Large Data Sets.

Graph Functions

The Big Data library supports graphing large data sets intelligently, using the following techniques to manage many thousands or millions of data points:

- Hexagonal binning. (That is, functions that create one point per observation in standard S-PLUS create a hexagonal binning plot when applied to a big data object.)
- Plot-specific summarizing. (That is, functions that are based on data summaries in standard S-PLUS compute the required summaries from a big data object.)
- Preprocessing data, using `table`, `tapply`, `loess`, or `aggregate`.
- Preprocessing using `interp` or `hist2d`.

Note

The Windows GUI editable graphics do not support big data objects. To use these graphics, create a data frame containing either all of the data or a sample of the data.

For a more detailed discussion of graph functions available in the Big Data library, see Chapter 5, Creating Graphical Displays of Large Data Sets.

Modeling Functions

Algorithms for large data sets are available for the following statistical modeling types:

- Linear regression.
- Generalized linear regression.
- Clustering.
- Principal components.

See the section Models on page 75 for more information about the modeling objects. See Table 3.7 for an overview of the big data modeling architecture.

If the data argument for a modeling function is a big data object, then S-PLUS calls the corresponding big data modeling function. The modeling function returns an object with the appropriate class, such as `bdLm`.

See Table A.12 in the Appendix for a list of the modeling functions that return a model object.

Generally, methods for a large data modeling class, such as `bdLm`, correspond to the methods for the standard modeling class, such as `lm`; however, the Big Data library supports a subset of the following:

- Modeling methods.
- Function arguments.
- Formulas.

If you request an unsupported option for a big data object, the algorithm stops with an error message. Reviewing the Big Data library modeling methods, functions, and formulas in the documentation can help avoid these errors.

Table 3.7: *Big Data library modeling architecture.*

Primary modeling function	Class
`glm`	`bdGlm`
`bdCluster`	`bdCluster`
`lm`	`bdLm`
`princomp`	`bdPrincomp`

See Tables A.10 through A.13 in the Appendix for lists of the functions available for large data set modeling. See the S-PLUS Language Reference for more information about these functions.

Formula operators

The Big Data library supports using the formula operators `+`, `-`, `*`, `:`, `%in%`, and `/`.

Time Classes

The following classes support time operations in the Big Data library. See the Appendix for more information.

Table 3.8: *Time classes.*

Class name	Comment
`bdSignalSeries`	A `bdSignalSeries` object from positions and data
`bdTimeDate`	A `bdVector` class
`bdTimeSeries`	See the section Time Series Operations for more information.
`bdTimeSpan`	A `bdVector` class

Time Series Operations

Time series operations are available through the `bdTimeSeries` class and its related functions. The `bdTimeSeries` class supports the same methods as the standard S-PLUS library's `timeSeries` class. See the S-PLUS Language Reference for more information about these classes.

Time and Date Operations

- When you create a time object using `timeSeq`, and you set the `bigdata` argument to `TRUE`, then a `bdTimeDate` object is created.
- When you create a time object using `timeDate` or `timeCalendar`, and any of the arguments are big data objects, then a `bdTimeDate` object is created.

See Table A.14 in the Appendix.

Note

`bdTimeDate` always assumes the time as Greenwich Mean Time (GMT); however, S-PLUS stores no time zone with an object. You can convert to a time zone with `timeZoneConvert`, or specify the zone in the `bdTimeDate` constructor.

Time Conversion Operations

To convert time and date values, apply the standard S-PLUS time conversion operations to the `bdTimeDate` object, as listed in Table A.14 in the Appendix.

Matrix Operations

The Big Data library does not contain separate equivalents to `matrix` and `data.frame`.

Standard S-PLUS matrix operations are available for `bdFrame` objects, including:

- matrix algebra (`+`, `-`, `/`, `*`, `!`, `&`, `|`, `>`, `<`, `==`, `!=`, `<=`, `=>`, `%%`, `%/%`)
- matrix multiplication (`%*%`)
- Crossproduct (`crossprod`)

(`solve` does not support big data objects in version 7.)

In algebraic operations, the operators require the big data objects to have appropriately-corresponding dimensions. Rows or columns are not automatically replicated.

Basic algebra

You can perform addition, subtraction, multiplication, division, logical (`!`, `&`, and `|`), and comparison (`>`, `<`, `=`, `!=`, `<=`, `>=`) operations between:

- A scalar and a `bdFrame`.
- Two `bdFrames` of the same dimension.
- A `bdFrame` and a single-row `bdFrame` with the same number of columns.
- A `bdFrame` and a single-column `bdFrame` with the same number of rows.

The library also offers support for elementwise `+`, `-`, `*`, `/`, and matrix multiplication (`%*%`).

Matrix multiplication is available for two `bdFrames` with the appropriate dimensions.

Cross Product Function

When applied against two `bdFrames`, the cross product function, `crossprod`, returns a `bdFrame` that is the cross product of the given `bdFrames`. That is, it returns the matrix product of the transpose of the first `bdFrame` with the second.

Summary

In this section, we've provided an overview to the Big Data library architecture, including the new data types, classes, and functions that support managing large data sets. For more detailed information and lists of functions that are included in the Big Data library, see the Appendix, Big Data Library Functions.

In the next chapter, Chapter 4, Exploring and Manipulating Large Data Sets, we provide examples for working with data sets using the types, classes, and functions described in this chapter.

EXPLORING AND MANIPULATING LARGE DATA SETS

4

Introduction **86**

Working in the S-PLUS Environment **87**
- Command-line functions 87
- Dialog box support 87
- Data Viewer 89

Manipulating Data: Census Example **91**
- Overview of Census Sample 91
- Overview of Data Manipulation Functions 92
- Work with the Census Example 93
- Displaying in a Simple Plot 98
- Displaying a Bar Plot 102
- Exporting Data 104
- Summary 104

Manipulating Data: Stock Sample **105**
- Preparing the Stock Sample Script 106
- Working with Time Series Data 109
- Summary 114

INTRODUCTION

This chapter includes information on the following topics for working with the S-PLUS Big Data library:

- Working from the command line.
- S-PLUS GUI support in Microsoft Windows: dialog boxes and data viewer.
- Manipulating data, demonstrated using census and stock examples.
- Creating graphs for large data sets.

WORKING IN THE S-PLUS ENVIRONMENT

When you use the Big Data library, you must perform all operations in the **Commands** window, except for importing and exporting data in the Windows environment. (The **Import Data**, **Select Data**, and **Export Data** dialog boxes accommodate big data objects.)

Command-line functions

Start the **Commands** window, and then type expressions and call big data functions at the command prompt. Remember that S-PLUS is *case sensitive*, and while many functions in the Big Data library are similar to standard S-PLUS functions, their case designation might be slightly different. For more information on the naming conventions in the Big Data library see the section The Big Data Library Architecture on page 69 of Chapter 3, The Big Data Library, and the Appendix, Big Data Library Functions. For more information about using the **Commands** window, see Chapter 10, Using the Commands Window in the *S-PLUS User's Guide*.

Dialog box support

The Big Data library provides dialog box support for the following two functions in Microsoft Windows only:

- `importData`
- `exportData`

For more information about importing and exporting data, including a list and descriptions of supported file types, see Chapter 5, Importing and Exporting Data in the *S-PLUS User's Guide*.

Import Data dialog box

If you are using Microsoft Windows, you can use the GUI dialog boxes for importing data. To import the data as a large data set using either the **Import From File** or **Import from Database** dialog boxes, select the **Import as Big Data** checkbox. For more

information about using the **Import Data** dialog box, in Windows, click **Help ▶ Available Help ▶ S-PLUS Help**, and then see the topic Importing Data Files.

Note

From the command line, import the data using the `importData` function. For more information on importing data from the command line, see the section Importing the Data on page 107.

S-PLUS 7 includes Census and Stock big data examples. The example files are installed in the samples directory in your S-PLUS program directory.

In the following section, import the census example data.

To import the Big Data census example data set using the S-PLUS GUI in Microsoft Windows

1. From the **File ▶ Import Data** menu, open the **Import from File** dialog box.
2. Under **File name,** click **Browse**, and in the **Select file to import** dialog box, browse to the census directory, by default located in your installation directory at **/samples/bigdata/census**.
3. Select **census.csv**.
4. In **File format**, select ASCII file - comma delimited (csv).
5. Select the **Import as Big Data** check box.
6. In the **Data set** text box, type P8.bd.
7. Click the **Options** tab.
8. Clear the **Strings as factors** check box.

Note

When you import data, you have the option to set the flag `stringsAsFactors` to `T` or `F` (the default is `T`). S-PLUS imposes a limit of 500 levels for `bdFactors`.

9. To preview the data, click the **Data Specs** tab, and then click **Update Preview**.

10. Click **OK** to import the data set.

Export Data dialog box

To export a large data set from S-PLUS using the S-PLUS GUI in Microsoft Windows, from the **File** menu, click **Export Data ▶ To File** or **Export Data ▶ To Database**.

Note
From the command line, export the data using the `exportData` function.

For a list of the data file types, in the S-PLUS for Windows GUI, click **Help ▶ Available Help ▶ S-PLUS Help**, and then in the Index, find the topic Export File Type.

To export the census example data set using the S-PLUS GUI in Microsoft Windows

1. From the **File ▶ Export Data** menu, open the **Export to File** dialog box.
2. For **Data frame,** provide the name of the data set (P8.bd).
3. For **File Name**, type **census.csv**.
4. For **Files of Type**, select ASCII file - comma delimited (csv).
5. Click **OK** to export the data set.

Data Viewer

The **Data Viewer** is a multi-page tabbed dialog box providing summaries of the different column types and a noneditable, scrollable grid view of the data. The **Data Viewer** is available only with the Enterprise Developer version of S-PLUS and requires that you have the Big Data library loaded. You can use the **Data Viewer** for both large data frames (`bdFrames`) and standard `data.frames`.

To view the example data set in the Data Viewer

You can display the data viewer from the **Commands** window.

- At the **Commands** window prompt, type:

```
bd.data.viewer(P8.bd)
```

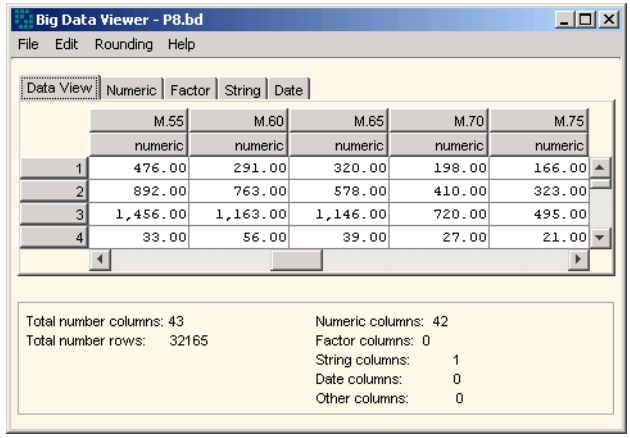

Big Data Viewer - P8.bd
File Edit Rounding Help
Data View Numeric Factor String Date
M.55 M.60 M.65 M.70 M.75
numeric numeric numeric numeric numeric
1 476.00 291.00 320.00 198.00 166.00
2 892.00 763.00 578.00 410.00 323.00
3 1,456.00 1,163.00 1,146.00 720.00 495.00
4 33.00 56.00 39.00 27.00 21.00
Total number columns: 43
Total number rows: 32165
Numeric columns: 42
Factor columns: 0
String columns: 1
Date columns: 0
Other columns: 0

Figure 4.1: *Data viewer displaying P8.bd.*

The **Data Viewer** contains the following tabs. The first tab (**Data View**) contains a table of the data. The remaining five contain summary information about the corresponding data type:

- **Data View**
- **Numeric**
- **Factor**
- **String**
- **Date**

In the **Data View** tab, you can scroll horizontally and vertically to examine the data. You can also change the size of the **Data Viewer** window to show more or less of the data table.

Note that the bottom pane of the **Data Viewer** provides summary information about the data set, including the numbers of rows and columns, and identifies the types of columns in the data set.

MANIPULATING DATA: CENSUS EXAMPLE

In this section, we begin manipulating the data in the P8.bd example used throughout the first half of this guide to demonstrate working with a large data set using the Big Data library functions. For practical reasons, this data set is not particularly large (about 33,000 rows and 40 columns); however, it is illustrative of a typical data set and the type of problem-solving users typically must perform.

Note

The entire sample script can be found in the default installation directory, in **samples/bigdata/census/census.demo.ssc**. You can work through the example demonstrations, below, or you can open the script and review it or run it.

After you import the data, your next task is to manipulate the data using standard and Big Data library functions.

Overview of Census Sample

The Big Data library Census sample reads in the pre-processed file, **census.csv**, which came from the Census Level-3 data. All data is binned by ZIP code tabulation area (ZCTA), using 5-digit zip codes, and includes information from the following census tables:

- **Table P8**: Contains the total ZCTA population data (P008001), with each column separated by gender and age, with the ages aggregated into 5-year bins (M.00, M.05, M.10... F.00, F.05, F.10, and so on). This table also includes the latitude (INTPTLAT) and longitude (INTPTLON) information for each ZCTA, which we use for plotting purposes.
- **Table H7**: Contains ZCTA tenancy information, including:
  - Total number of occupied housing units (H007001).
  - number of owned homes (H007002).
  - number of rented homes (H007003).

Overview of Data Manipulation Functions

The table below lists some common tasks for working with large data objects. Corresponding to the tasks is a list of functions that apply to the task. Each function is described in further detail, with an example showing how to use it, in its corresponding help topic, which you can access easily from the command line by typing `help(functionname)`.

The tasks that apply to the census data set are described in more detail, with procedures and example code, later in this chapter.

Table 4.1: *Data manipulation tasks and their associated functions*

Task	Function names
Importing data	`importData`
Converting data (for example, from big data to a data frame)	`bd.coerce`
Generating a vector of random numbers	`rbeta`, `rbinom`, `rcauchy`, `rchisq`, `rep`, `rexp`, `rf`, `rgamma`, `rgeom`, `rhyper`, `rlnorm`, `rlogis`, `rmvnorm`, `rnbinom`, `rnorm`, `rnrange`, `rpois`, `rstab`, `rt`, `runif`, `rweibull`, `rwilcox`, `seq`
Displaying and exploring `bdFrame` data	`bd.cor`, `bd.crosstabs`, `bd.univariate`, `show`, `summary`, `bd.data.viewer`
Manipulating data in blocks	`bd.block.apply`, `bd.by.group`, `bd.by.window`
Manipulating time series data	`print`, `summary`, `aggregateSeries`, `align`, `diff`, `seriesMerge`
Cleaning existing data	`bd.remove.missing`, `bd.normalize`, `bd.duplicated`, `bd.unique`
Splitting data	`bd.split`, `bd.split.by.group`, `bd.split.by.window`
Appending data sets (either by rows or by columns)	`bd.append`, `bd.join`

Table 4.1: *Data manipulation tasks and their associated functions*

Task	Function names
Manipulating rows	`bd.filter.rows, bd.partition, bd.relational.restrict, bd.sample, bd.select.rows, bd.shuffle, bd.sort, rowMaxs, rowMeans, rowMins, rowRanges, rowStdevs, rowSums, rowVars`
Manipulating columns	`bd.aggregate, bd.bin, bd.create.columns, bd.filter.columns, bd.modify.columns, bd.relational.divide, bd.relational.project, bd.reorder.columns, bd.transpose, bd.stack, bd.unstack, colMaxs, colMeans, colMins, colRanges, colStdevs, colSums, colVars`
Exporting data	`exportData`
Relational operations	`bd.relational.difference, bd.relational.intersection, bd.relational.join, bd.relational.product, bd.relational.union`
Identifying and removing orphan caches	`bd.cache.cleanup, bd.cache.info`
Store and retrieve objects	`bd.pack.object, bd.unpack.object`

Work with the Census Example

In the following exercises, import, filter, and manipulate the Census data.

Importing Existing Data

This section describes importing the example data set from a data source using the `importData` command in the **Commands** window. For more information about `importData`, see its help topic.

To import the data set

1. In the **Commands** window, type:

```
P8.bd<-importData(paste(getenv("SHOME"),
    "/samples/bigdata/census/census.csv", sep=""),
    stringsAsFactors=F, startRow=1, bigdata=T)
```

Note

When you import data, you have the option to set the flag `stringsAsFactors` to `T` or `F` (the default is `T`). S-PLUS imposes a limit of 500 levels for `bdFactors`.

2. Display the resulting data set in the data viewer:

```
bd.data.viewer(P8.bd)
```

Each cell in the rectangular big data object displayed in the Viewer contains the count of either males or females within 5-year age bins, shown for each ZCTA. Each ZCTA is shown as a separate row; each male or female age bin is shown as a separate column.

The columns labeled M00, M05, M10, and so on, represent the number of males from 0 to 4 years, 4 to 9 years, 10 to 14 years. The columns labeled F00, F05, F10, and so on, represent the number of females in those age groups. The last bin contains males or females age 85 and older.

For example, the first ZCTA shown is 00601, and there are 712 males from 0-4 years old in this ZCTA.

Although these raw counts are interesting, in their present form, the data for two ZCTAs cannot be compared directly, because the total populations in the ZCTAs vary greatly.

The objective in this section is to demonstrate using several big data functions by manipulating this data set.

Loading Supporting Source Files

The census example uses some customized functions. To continue working with the example, provide references to these source files.

To reference the supporting function files for the example

1. Open the **Commands** window.
2. At the command prompt, type

```
source(paste(getenv("SHOME"), "/samples/bigdata/
    census/my.vbar.q", sep=""))
source(paste(getenv("SHOME"), "/samples/bigdata/
    census/graph.setup.q", sep=""))
```

Note

`graph.setup.q` runs `graphsheet` on the Windows platform and `java.graph` on Unix platforms. If you are working with this example in the S-PLUS Workbench, remember that Eclipse does not work with `java.graph` on the Unix platform.

Converting Data

You can convert a standard data frame object to a `bdFrame` object. In the following procedure, you can load the census data set as a standard S-PLUS data frame, and then convert it to a `bdFrame`. (Later, you can convert the `bdFrame` to a `data.frame`.)

Note

The steps in this section simply demonstrate converting big data; it is not required for the remainder of the example. You have already imported the data as a big data object in the earlier section To import the data set.

To read census data as a data frame and then convert to a bdFrame

1. Load the Big Data library, if it is not already loaded.
2. Read the census data without setting the `bigdata=T` argument.

   ```
   small.Census <-
       importData(paste(getenv("SHOME"),"/samples/
       bigdata/census/census.csv", sep=""),
        type="ASCII", stringsAsFactors=F)

   larger.Census<-as.bdFrame(small.Census, bigdata=T)
   ```

3. View the resulting data in the Data Viewer. In the Commands window, type

   ```
   bd.data.viewer(larger.Census)
   ```

Likewise, you can convert an S-PLUS vector to a `bdVector`.

To convert between an S-PLUS vector and a bdVector subclass

1. In the **Commands** window, type

```
ZCTA.bv <- as.bdCharacter(P8.bd$ZCTA5)
```

Note

While you can use either `bd.coerce` or functions like `as.bdCharacter` and `as.bdFrame` to convert standard objects to big data objects, you must use `bd.coerce` to convert big data objects to standard objects. This technique provides a single function to convert big data objects to standard data objects so it is easier to track where big data is coerced to standard, and to make it easier for you to write code that scales to handle arbitrarily large data.

2. To view the ZIP codes as strings, type:

   ```
   ZCTA.bv
   ```

   To view the `bdVector` data in the data viewer, type

   ```
   bd.data.viewer(ZCTA.bv)
   ```

   The ZCTAs are stored as strings in this table; click the **Strings** tab in the **Data Viewer** to see the results.

3. Next, you can coerce the ZCTA data to numbers:

   ```
   Zip.Code.Tab.Areas.Num.bv<-as.bdNumeric(P8.bd$ZCTA5)
   ```

   Examine the results in the **Data Viewer**, if you choose.

Note

ZIP codes are best imported as character strings; otherwise, S-PLUS truncates the leading 0 for east coast ZIP codes (e.g., “02139” becomes “2139”.)

Manipulating Rows

When you examined the sample census data in the **Data Viewer**, you might have seen that the data set contains several rows of uninformative data: rows showing ZCTAs containing letters and the population bins all showing 0. In the following exercise, examine and filter those rows, and then re-display the data set in the Data Viewer.

To filter the rows

1. To only keep rows where P008001 is greater than 0:

```
P8.bd <- bd.filter.rows(P8.bd,
   expr="P008001>0")
```

`bd.filter.rows` has a logical argument `include` with a default of `TRUE`. If you add `include=F` to the above call, you would drop all rows where P008001 is greater than 0.

2. Show the data set in the Data Viewer.

   ```
   bd.data.viewer(P8.bd)
   ```

   Note that the data set now contains 32,165 rows. The filtering function removed 1,013 rows containing uninteresting data.

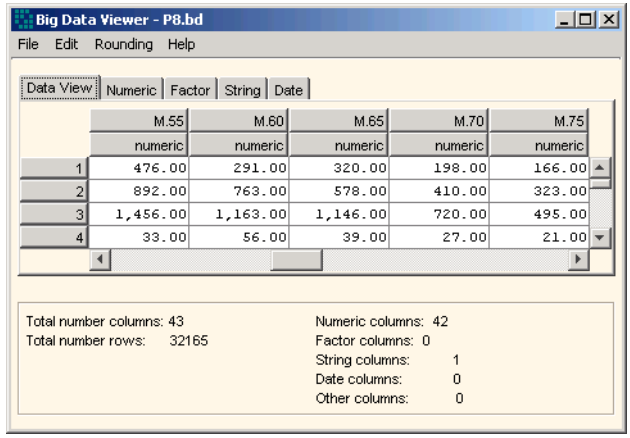

Big Data Viewer - P8.bd
File Edit Rounding Help
Data View Numeric Factor String Date
M.55 M.60 M.65 M.70 M.75
numeric numeric numeric numeric numeric
1 476.00 291.00 320.00 198.00 166.00
2 892.00 763.00 578.00 410.00 323.00
3 1,456.00 1,163.00 1,146.00 720.00 495.00
4 33.00 56.00 39.00 27.00 21.00
Total number columns: 43
Total number rows: 32165
Numeric columns: 42
Factor columns: 0
String columns: 1
Date columns: 0
Other columns: 0

Figure 4.2: *P8.bd*

Other functions that provide cleaning, filtering, and compiling are:

`bd.partition.rows`, `bd.sample.rows`, `bd.select.rows`, `bd.shuffle.rows`, `bd.sort.rows`.

Sorting and Manipulating the Data

As part of your data manipulation, you can separate the data set, according to the types of data in its columns, add reference columns, and add columns representing values manipulated to provide more usable information.

To create reference and data columns:

- Create separate data sets to hold the reference data columns and the data columns, assigning the gender and age bins to the object.

```
P8.ref.bd <- P8.bd[,c(1:4, 41:43)]  # ref cols
P8.data.bd <- P8.bd[,5:40] # data cols
```

  - The reference data columns contain the ZCTA information, the population totals, and the housing information.
  - The data columns contain the gender and age bins.

To create and transform columns of existing data

1. Add to the reference data set columns containing the adjusted scale of latitude and longitude ("Lat" and "Lon"), and assign the resulting data set to `P8.suppl.bd`. The original latitude and longitude values (`INTPTLAT` and `INTPLTLON`) were stored as large integer values.

```
P8.suppl.bd <- bd.create.columns(P8.ref.bd,
    exprs=c("INTPTLAT/1.e6", "INTPTLON/1.e6"),
    names=c("Lat","Lon"),
    types="continuous",
    copy=T)
```

   (In the next section, plot the ZIP code distribution to examine its density.)

2. Open the data viewer and examine the latitude and longitude variables.

```
bd.data.viewer(P8.suppl.bd)
```

Displaying in a Simple Plot

In this exercise, use the data set `P8.suppl.bd` with the adjusted latitude and longitude values to display the distribution of zip code locations in a simple hexbin plot.

This simple plot maps the density of zip code locations in the United States and Puerto Rico.

To display zip code density

1. Next, create new `Lat` and `Lon` variables on the correct scale, and then save along with the original reference data in a new data set, `p8.suppl.bd`. In the **Commands** window, type

   ```
   plot(P8.suppl.bd$Lon,P8.suppl.bd$Lat)
   ```

   Note that the `plot` function produces the `hexbin` plot by default for big data objects, rather than a scatter plot.

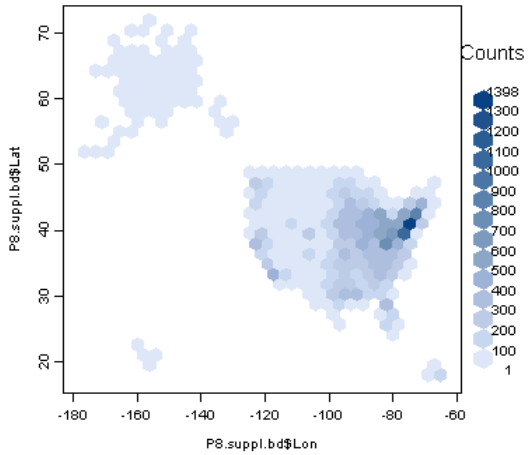


Figure 4.3: *ZIP code concentration plot.*

2. Examine the plot and notice the concentration of ZIP codes in the Northeast, Ohio valley, and upper mid-west, along with a relatively smaller concentration on the California coast and other urban population centers.

For more information about the graph functions available for large data sets, see Chapter 5, Creating Graphical Displays of Large Data Sets.

Transforming the Data

To compare the distribution of age/gender groups across different ZCTAs, you must adjust the values for the total population count within the ZCTA. The simplest adjustment is to divide each age/gender population value by the total population for that ZCTA. This procedure yields the fraction of the population for that ZCTA in each

age/gender group. This transformation makes column comparisons meaningful when you do a cluster analysis in Chapter 6, Modeling Large Data Sets.

To transform the data

1. Divide each of the data columns by the total population for each row in the reference data set (which is contained in the column named "P008001"), and then store this transformed data in a new big data object.

   ```
   P8.dataN.bd <- P8.data.bd/P8.ref.bd[,"P008001"]
   ```

2. Modify this new object by appending an `"N"` to the column labels to signify they've been normalized. Both the name of the new big data object and its variables contain "N".

   ```
   names(P8.dataN.bd) <- paste(names(P8.dataN.bd),
      "N",sep="")
   ```

   Alternatively, you can use the `bd.modify.columns` function:

   ```
   P8.dataN.bd <- bd.modify.columns(P8.dataN.bd,
       names(P8.dataN.bd), paste(names(P8.data.bd),
      "N", sep=""))
   ```

   You can use `bd.modify.columns` for more extensive column manipulation, such as changing column types and identifying columns to keep or drop, as well as changing column names. For more information about `bd.modify.columns`, see its help topic.

3. Display the resulting normalized data set.

   ```
   bd.data.viewer(P8.dataN.bd)
   ```

   Note that the values are no longer integer counts, but fractions between 0 and 1.

To transform by average per bin

You can now directly compare the transformed data P8.dataN.bd across all 32,165 ZCTAs. We use clustering methods to seek geographic patterns of interesting groups of populations. Before proceeding to the clustering step, though, perform one further transformation of the data.

We want a factor of 2 change in population to be as significant in the 80 year bin (a very small bin) as it is in the 30 year bin (a very large bin). Just as you adjusted for differing populations across ZCTAs, now adjust for the differing numbers across age/gender groups.

1. Calculate the mean for each age/gender group column.

   ```
   P8.dataN.mean <- colMeans(P8.dataN.bd)
   ```

2. Create new series of columns by dividing by this national average value per group. The resulting object contains the national average demographic profile in these age/gender groups.

   The `bd.create.columns` function accepts values for the new columns and the expressions to form them. It is often convenient to pre-form these character vectors before the actual call, as shown here.

   ```
   column.exprs <- paste( names(P8.dataN.bd),
      paste("/P8.dataN.mean[", 1:36, "]", sep=""),
      sep="" )
   column.names.N <- names(P8.dataN.bd)
   column.names.Nz <- paste( column.names.N,
      "z", sep="" )
   P8.dataNz.bd <- bd.create.columns(
      P8.dataN.bd,
      exprs=column.exprs,
      names=column.names.Nz,
      row.language=F)
   ```

Note

The `row.language` argument above is set to `F` because the expressions contain the subset operator `[`, which requires S-PLUS in its evaluation.

3. Display the new data in the **Data Viewer**. Note that the variable has a z appended to indicate that this is the normalized data.

   ```
   bd.data.viewer(P8.dataNz.bd) # 32,165 rows
   ```

This table shows the ratio of the population for each group compared to the national average. For example, the value of M05 in ZIP code 07043 is 1.2, meaning that this region has proportionally 20% more males in this age group than the national average.

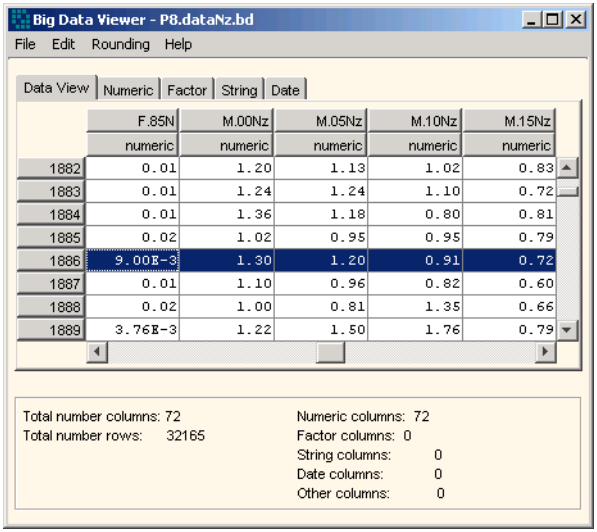

Big Data Viewer - P8.dataNz.bd
File Edit Rounding Help
Data View Numeric Factor String Date
F.85N M.00Nz M.05Nz M.10Nz M.15Nz
numeric numeric numeric numeric numeric
1882 0.01 1.20 1.13 1.02 0.83
1883 0.01 1.24 1.24 1.10 0.72
1884 0.01 1.36 1.18 0.80 0.81
1885 0.02 1.02 0.95 0.95 0.79
1886 9.00E-3 1.30 1.20 0.91 0.72
1887 0.01 1.10 0.96 0.82 0.60
1888 0.02 1.00 0.81 1.35 0.66
1889 3.76E-3 1.22 1.50 1.76 0.79
Total number columns: 72
Total number rows: 32165
Numeric columns: 72
Factor columns: 0
String columns: 0
Date columns: 0
Other columns: 0

Figure 4.4: *P8.data.Nz.bd*

Displaying a Bar Plot

In this exercise, using the normalized data from the section Transforming the Data on page 99, produce a single bar plot to show the national average of female and male age distributions for the whole population.

This bar plot shows females to the left of 0 and males to the right.

To display the gender bar plot

1. In the Commands window, type

```
barplot(rbind(P8.dataN.mean[1:18], -P8.dataN.mean
   [19:36]), horiz=T)
```

2. Examine the plot and notice the baby boom ages and the subsequent boomlet. Also note the difference in population between genders at greater ages.

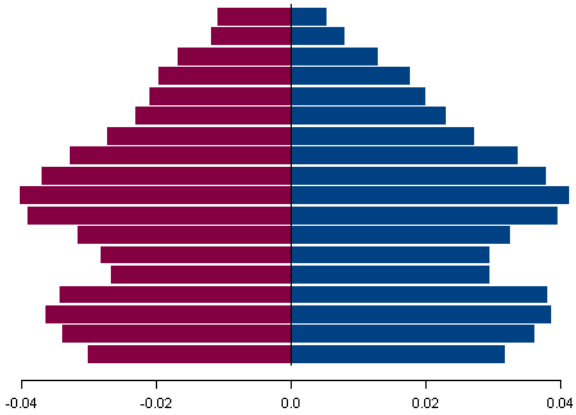


Figure 4.5: *Bar plot of age and gender data.*

For more information about the graph functions available for large data sets, see Chapter 5, Creating Graphical Displays of Large Data Sets.

Joining Columns

In the course of our data processing, the data and the geographic information have become separated. In this exercise, join the normalized data row-by-row, with the informational columns.

To join columns

1. In this step, combine the transformed gender and age data set (`P8.dataNz.bd`) with the latitude and longitude data set (`P8.suppl.bd`) to get one data set. (Later, using this combined data set, you can plot gender and age information on a map.) In the **Commands** window, type

```
P8.Nz.bd <- bd.join(list(P8.suppl.bd,
   P8.dataNz.bd) )
```

2. Display the results in the **Data Viewer**. Note the latitude and longitude variables.

```
bd.data.viewer(P8.Nz.bd)
```

Exporting Data

This optional step just demonstrates exporting data to an ASCII text file. Optionally, skip this step and continue to the next chapter.

- In the **Commands** window, type

```
exportData(P8.bd, file="exportedfile.txt",
    type="ASCII")
```

These options indicate that the data set is exported as an ASCII text file. The file name and location are specified by `file`. See the help file for `exportData` for a description of all options for exporting to a database.

Summary

The next steps in working with the Census example are to perform cluster modeling. These steps and discussion are continued in Chapter 6, Modeling Large Data Sets.

The next section in this chapter provides further practice importing, manipulating, and plotting time series data for a sample financial data set.

MANIPULATING DATA: STOCK SAMPLE

In this section, we work with a different data set, a financial data set, using the script **stock.ssc** and associated **.csv** files, provided in the default S-PLUS Installation sample directory. Again, for practical reasons, this data set is not particularly large (26 columns, 2729 rows); however, it illustrates features and tasks of working with a typical large data set that contains financial data, including time series information and missing data. This example can easily be run with a data set of millions of rows without requiring additional RAM.

In this stock analysis example, you will:

- Manipulate the data (join, filter, remove missing data, create columns, and so on)
- Create a time series object
- Plot the time series
- Use different methods to analyze the betas using linear modeling.
- Compare the analysis methods.

Note

The entire sample script can be found in the default installation directory **/samples/bigdata/stocks/stock.ssc**. You can work through the example demonstrations, below, or you can open the script and review it or run it.

Preparing the Stock Sample Script

This example examines the daily close prices of 24 conglomerate stocks and the S&P 500 index from 01/01/1994 to 11/01/2004.

Table 4.2: *Stock Data used in Example.*

Stock symbol	Company name
cbe	Cooper Industries
cr	Crane Company
dov	Dover Corporation
fo	Fortune Brands, Incorporated
ge	General Electric Company
gy	GenCorp, Incorporated
hon	Honeywell International
hsc	Harsco Corporation
kor	Koor Industries LTD
kt	Katy Industries, Incorporated
lgl	Lynch Corporation
mitsy	Mitsui & Co. LTD
mmm	3M Company
ppg	PPG Industries, Incorporated
quix	Quixote Corporation
rok	Rockwell Automation

Table 4.2: *Stock Data used in Example. (Continued)*

Stock symbol	Company name
rtk	Rentech, Incorporated
sxi	Standex International Corporation
tfx	Teleflex, Incorporated
tmo	Thermo Electron Corporation
tvin	TVI Corporation
txt	Textron, Incorporated
tyc	Tyco International LTD
utx	United Technologies Corporation

Importing the Data

This example contains 25 .csv files: one for each of the represented 24 conglomerate stocks and one for the S&P 500 index. First, specify an object to contain the stock IDs, and then import each of the 24 conglomerate stock files

Prepare the conglomerate stock data

1. Specify the constituent stock IDs and assign them to the object `stockNames`.

```
stockNames <- c("cbe", "cr", "dov", "fo",
    "ge", "gy", "hon", "hsc","kor", "kt",
    "lgl", "mitsy", "mmm", "ppg", "quix",
    "rok", "rtk", "sxi", "tfx", "tmo",
    "tvin", "txt", "tyc", "utx")
```

2. Import the corresponding source files.

```
srcFileNames <- paste(getenv("SHOME"),
    "/samples/bigdata/stocks/",
       paste(stockNames, ".csv", sep=""),
       sep="")
```

Manipulating the Stock Data

In this section, create a list of close price series for the stocks. If you are working with a large number of stocks, this list object is potentially quite large; however, when the expression is evaluated, the component `bdFrame` objects are not loaded into virtual memory.

To create a list of close price series

1. Read close price series for the stocks from file sources.

```
closePricesList <-
    lapply(srcFileNames, function(fileName) {
    importData(fileName, keep=c("DATE", "CLOSE"),
    bigdata=T)})

names(closePricesList) <- casefold(stockNames,
    upper=T)
```

2. Combine the close columns into one data set. Note that this function works even if the series items do not all have the same date column.

```
closePrices.bd <- bd.join(closePricesList,
    key.columns="DATE", suffixes=paste(".",
    names(closePricesList), sep=""))
```

3. Remove the `"CLOSE"` column name markers.

```
colIds(closePrices.bd) <-
    substituteString("CLOSE\.", "",
    colIds(closePrices.bd))
```

Importing the S&P 500 Index Data

The data for the S&P 500 Index is drawn from the same date range, 01/01/1994 to 11/01/2004.

To import and Join the S&P 500 Index data

1. Read close price series for the S&P 500 Index from the index data file (**inx.csv**).

```
closeSP500.bd <- importData(paste(getenv("SHOME"),
    "samples","bigdata","stocks","inx.csv",
    sep=dirSeparator()), keep=c("DATE", "CLOSE"),
    bigdata=T)
```

2. Edit the S&P 500 Index column names to identify the column as S&P 500 data.

```
names(closeSP500.bd)[-1] <- "SP500"
```

3. Join the index series with those of the conglomerate stocks.

```
closePrices.bd <- bd.join(list(closePrices.bd,
   closeSP500.bd), key.columns="DATE")
```

4. View the univariate summaries.

```
summary(closePrices.bd)
```

Cleaning the Stock Data

When you examine `closePrices.bd`, notice that most of the stocks, as well as the S&P 500 Index, have 97 NA values. These NA values represent the days the market was closed for holidays over the 10+ year period. In the next steps, drop these NA values.

To drop the NA values

1. Identify the missing days for the entire index and remove those days represented by NAs in the S&P 500 Index.

```
closePrices.bd <-
   bd.remove.missing(closePrices.bd, columns="SP500",
   method="drop")
```

2. Examine the whole data set in the Data Viewer.

```
bd.data.viewer(closePrices.bd)
```

Of the 24 stocks, notice that only RTK, TVIN, KOR, MITSY still have NAs. (These constituents were not listed in the S&P 500 for the entire observation period.)

Working with Time Series Data

In the following steps, using the stock sample data, remove the shorter-term constituents, and then create a time series representing the conglomerate stock closing price returns.

Creating the Time Series

In the previous section, you discovered that the stocks RTK, TVIN, KOR, MITSY have a shorter history than the other stocks. In this analysis, consider only the constituents with the 10+ years of history in our date range.

In this section, remove the shorter-term constituents and create the time series of returns, and then compute the daily returns time series.

To create the time series

1. Create a `bdTimeSeries` object, removing the stock IDs that do not have the entire history.

```
keepIds <- !is.element(colIds(closePrices.bd),
   c("DATE", "RTK", "TVIN", "KOR", "MITSY"))
closePrices.ts <-
   bdTimeSeries(data=closePrices.bd[, keepIds],
   positions=closePrices.bd[, "DATE"])
print(class(closePrices.ts))
```

2. Compute daily returns time series and assign it to the object `dailyReturns.ts`.

```
dailyReturns.ts <- diff(log(closePrices.ts))
```

Plotting the Time Series

In this section, create a time series object of the cumulative returns and then plot them. Add a label to each series.

To plot the cumulative returns

1. Create a time series object of the cumulative returns

```
cumulativeReturns.ts <- cumsum(dailyReturns.ts)
```

2. Plot the cumulative returns.

```
plot(cumulativeReturns.ts,
   main="Cumulative returns of SP500 Index
   and 20 Stocks", ylab="Returns")
```

3. Annotate each series.

```
lastObs <- positions(cumulativeReturns.ts) ==
    max(positions(cumulativeReturns.ts))
text(rep(1, numCols(dailyReturns.ts)),
   unlist(seriesData(cumulativeReturns.ts)
   [lastObs, , drop=T]),colIds(dailyReturns.ts),
   col=3, cex=0.5)
```

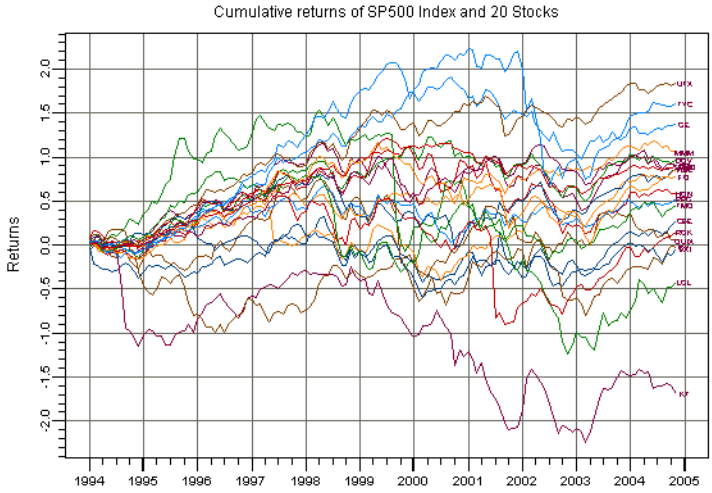


Figure 4.6: *Plot of cumulative returns.*

Analyzing the betas

The beta is one way of measuring how returns on an asset change when the market changes. In this example, the market is represented by the S&P 500 Index. This analysis shows two separate techniques for analyzing the betas. The second technique, Approach 2, is slightly faster.

To calculate betas using Approach #1a

1. Capture column IDs of the stocks.

```
constituentNames <-
    colIds(dailyReturns.ts)[colIds(dailyReturns.ts)
    != "SP500"]
```

2. Set the process time for calculating the betas using this approach.

```
t0 <- proc.time()[3]
```

3. Initialize the vector of betas.

```
betas1.a <-
    structure(numeric(length(constituentNames)),
    names=constituentNames)
```

4. Loop through the stocks, and calculate the beta directly as a regression coefficient.

```
for (constituentName in constituentNames){
       lmFormula <- paste(constituentName, "~ SP500")
       beta <- lm(lmFormula,
       data=dailyReturns.ts@data)
       betas1.a[constituentName] <- coef(beta)[2]
   }
timeBetas1.a <- proc.time()[3] - t0
```

To calculate betas using Approach #1b

1. Set the time to calculate the betas using this approach.

```
t0 <- proc.time()[3]
```

2. Initialize the vector of betas.

```
betas1.b <-
   structure(numeric(length(constituentNames)),
   names=constituentNames)
```

3. Loop through the stocks, and calculate the beta directly as a regression coefficient.

```
for (constituentName in constituentNames){
       beta <- lsfit(seriesData(dailyReturns.ts)
       [, "SP500"],
       seriesData(dailyReturns.ts)[, constituentName])
       betas1.b[constituentName] <- beta$coef[2]
   }
timeBetas1.b <- proc.time()[3] - t0
print(all.equal(betas1.a, betas1.b))
```

To calculate betas using Approach #2

1. Set the time to calculate the betas using this approach.

```
t0 <- proc.time()[3]

stdevs <- colStdevs(dailyReturns.ts)
```

2. Calculate betas without an explicit loop, by adjusting the correlation coefficients.

```
corSP500.bd <- bd.cor(seriesData(dailyReturns.ts),
   y.columns=constituentNames, x.columns="SP500")
betas2 <- unlist(corSP500.bd[1, -1, drop=T]) *
   stdevs[constituentNames] / stdevs["SP500"]
timeBetas2 <- proc.time()[3] - t0
```

Comparing techniques

Compare the answers from Approaches 1 and 2.

To check both techniques

1. Examine both betas

```
print(all.equal(betas1.b, betas2))
```

Plot the beta

Plot the 10-year return against the beta calculated over that period.

To plot the beta

1. Create an object for the 10-year return.

```
tenyrReturn <-
   unlist(seriesData(cumulativeReturns.ts)[lastObs,
   constituentNames, drop=T])
```

2. Plot the 10-year return.

```
plot(betas2, tenyrReturn,
   main="10-yr Return vs. Beta",
   xlab="beta", ylab="return", pch=16)

text(betas2 + 0.015, tenyrReturn, constituentNames,
   cex=0.7, adj=0)

points(1, seriesData(cumulativeReturns.ts)[lastObs,
   "SP500"], pch=18, col=3)

text(1 + 0.015,
   seriesData(cumulativeReturns.ts)[lastObs,
   "SP500"], "SP500", cex=0.7, col=3, adj=0)
```

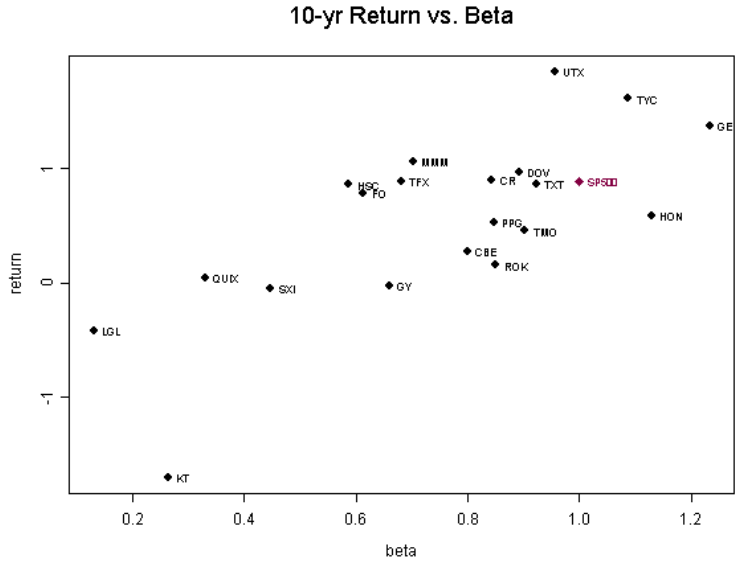


Figure 4.7: *Plot of 10-year return vs. the beta.*

Summary

In this chapter, you practiced exploring and manipulating big data sets using common Big Data library functions, including:

- Importing and viewing data.
- Coercing data to a smaller data set and back to a big data set.
- Sorting and filtering data.
- Creating columns.
- Appending data sets.
- Joining rows.
- Transforming data.
- Rendering graphs.
- Comparing calculation techniques.
- Plotting a time series.

In the next chapter, review the graph and chart functions that the Big Data library supports, using small, stand-alone data examples to call each graph function and display a different graph or chart type.

CREATING GRAPHICAL DISPLAYS OF LARGE DATA SETS

5

Introduction **116**

Overview of Graph Functions **117**
Functions Supporting Graphs 117

Example Graphs **123**
Plotting Using Hexagonal Binning 123
Adding Reference Lines 128
Plotting by Summarizing Data 133
Creating Graphs with Preprocessing Functions 144
Unsupported Functions 157

INTRODUCTION

This chapter includes information on the following:

- An overview of the graph functions available in the Big Data Library, listed according to whether they take a big data object directly, or require a preprocessing function to produce a chart.
- Procedures for creating plots, traditional graphs, and Trellis graphs.

Note

In Microsoft Windows, editable graphs in the graphical user interface (GUI) do not support big data objects. To use these graphs, create an S-Plus `data.frame` containing either all of the data or a sample of the data.

OVERVIEW OF GRAPH FUNCTIONS

The Big Data Library supports most (but not all) of the traditional and Trellis graph functions available in the S-PLUS library. The design of graph support for big data can be attributed to practical application. For example, if you had a data set of a million rows or tens of thousands of columns, a cloud chart would produce an illegible plot.

Functions Supporting Graphs

This section lists the functions that produce graphs for big data objects. If you are unfamiliar with plotting and graph functions in S-PLUS, review the following chapters in the *Application Developer's Guide*:

- Chapter 1, Editable Graphics Commands
- Chapter 2, Traditional Graphics
- Chapter 3, Traditional Trellis Graphics

Implementing plotting and graph functions to support large data sets requires an intelligent way to handle thousands of data points. To address this need, the graph functions to support big data are designed in the following categories:

- Functions to plot big data objects without preprocessing, including:
  - Functions to plot big data objects by hexagonal binning.
  - Functions to plot big data objects by summarizing data in a plot-specific manner.
- Functions providing the preprocessing support for plotting big data objects.
- Functions requiring preprocessing support to plot big data objects.

The following sections list the functions, organized into these categories. For an alphabetical list of graph functions supporting big data objects, see the Appendix, Big Data Library Functions.

Using `cloud` or `parallel` results in an error message. Instead, sample or aggregate the data to create a `data.frame` that can be plotted using these functions.

Graph Functions using Hexagonal Binning

The following functions can plot a large data set (that is, can accept a big data object without preprocessing) by plotting large amounts of data using hexagonal binning.

Table 5.1: *Functions for plotting big data using hexagonal binning.*

Function	Comment
`pairs`	Can accept a `bdFrame` object.
`plot`	Can accept a `hexbin`, a single `bdVector`, two `bdVectors`, or a `bdFrame` object.
`splom`	Creates a Trellis graphic object of a scatterplot matrix.
`xyplot`	Creates a Trellis graphic object, which graphs one set of numerical values on a vertical scale against another set of numerical values on a horizontal scale.

Functions Adding Reference Lines to Plots

The following functions add reference lines to hexbin plots.

Table 5.2: *Functions that add reference lines to hexbin plots.*

Function	Type of line
`abline(lsfit())`	Regression line.
`lines(loess.smooth())`	Loess smoother.
`lines(smooth.spline())`	Smoothing spline.
`panel.lmline`	Adds a least squares line to an `xyplot` in a Trellis graph.

Table 5.2: *Functions that add reference lines to hexbin plots. (Continued)*

Function	Type of line
`panel.loess`	Adds a loess smoother to an `xyplot` in a Trellis graph.
`qqline()`	QQ-plot reference line.
`xyplot(lmline=T)`	Adds a least squares line to an `xyplot` in a Trellis graph.

Graph Functions Summarizing Data

The following functions summarize data in a plot-specific manner to plot big data objects.

Table 5.3: *Functions that summarize in plot-specific manner.*

Function	Description
`boxplot`	Produces side by side boxplots from a number of vectors. The boxplots can be made to display the variability of the median, and can have variable widths to represent differences in sample size.
`bwplot`	Produces a box and whisker Trellis graph, which you can use to compare the distributions of several data sets.
`plot(density)`	`density` returns x and y coordinates of a non-parametric estimate of the probability density of the data.
`densityplot`	Produces a Trellis graph demonstrating the distribution of a single set of data.
`hist`	Creates a histogram.
`histogram`	Creates a histogram in a Trellis graph.
`qq`	Creates a Trellis graphic object comparing the distributions of two sets of data

Table 5.3: *Functions that summarize in plot-specific manner. (Continued)*

Function	Description
`qqmath`	Creates normal probability plot for only one data object in a Trellis graph. `qqmath` can also make probability plots for other distributions. It has an argument distribution whose input is any function that computes quantiles.
`qqnorm`	Creates normal probability plot in a Trellis graph. `qqnorm` can accept a single `bdVector` object.
`qqplot`	Creates normal probability plot in a Trellis graph. Can accept two `bdVector` objects. In `qqplot`, each vector or `bdVector` is taken as a sample, for the x- and y-axis values of an empirical probability plot.
`stripplot`	Creates a Trellis graphic object similar to a box plot in layout; however, it displays the density of the datapoints as shaded boxes.

Functions Providing Support to Preprocess Data for Graphing

The following functions are used to preprocess large data sets for graphing:

Table 5.4: *Functions used for preprocessing large data sets.*

Function	Description
`aggregate`	Splits up data by time period or other factors and computes summary for each subset.
`hexbin`	Creates an object of class hexbin. Its basic components are a cell identifier and a count of the points falling into each occupied cell.
`hist2d`	Returns a structure for a 2-dimensional histogram which can be given to a graphics function such as `image` or `persp`.
`interp`	Interpolates the value of the third variable onto an evenly spaced grid of the first two variables.

Table 5.4: *Functions used for preprocessing large data sets. (Continued)*

Function	Description
`loess`	Fits a local regression model.
`loess.smooth`	Returns a list of values at which the loess curve is evaluated.
`lsfit`	Fits a (weighted) least squares multivariate regression.
`smooth.spline`	Fits a cubic B-spline smooth to the input data.
`table`	Returns a contingency table (array) with the same number of dimensions as arguments given.
`tapply`	Partitions a vector according to one or more categorical indices.

Functions Requiring Preprocessing Support for Graphing

The following functions do not accept a big data object directly to create a graph; rather, they require one of the specified preprocessing functions.

Table 5.5: *Functions requiring preprocessors for graphing large data sets.*

Function	Preprocessors	Description
`barchart`	`table`, `tapply`, `aggregate`	Creates a bar chart in a Trellis graph.
`barplot`	`table`, `tapply`, `aggregate`	Creates a bar graph.
`contour`	`interp`, `hist2d`	Make a contour plot and possibly return coordinates of contour lines.
`contourplot`	`loess`	Displays contour plots and level plots in a Trellis graph.

Table 5.5: *Functions requiring preprocessors for graphing large data sets. (Continued)*

Function	Preprocessors	Description
`dotchart`	`table`, `tapply`, `aggregate`	Plots a dot chart from a vector.
`dotplot`	`table`, `tapply`, `aggregate`	Creates a Trellis graph, displaying dots and labels.
`image`	`interp`, `hist2d`	Creates an image, under some graphics devices, of shades of gray or colors that represent a third dimension.
`levelplot`	`loess`	Displays a level plot in a Trellis graph.
`persp`	`interp`, `hist2d`	Creates a perspective plot, given a matrix that represents heights on an evenly spaced grid.
`pie`	`table`, `tapply`, `aggregate`	Creates a pie chart from a vector of data.
`piechart`	`table`, `tapply`, `aggregate`	Creates a pie chart in a Trellis graph
`wireframe`	`loess`	Displays a three-dimensional wireframe plot in a Trellis graph.

EXAMPLE GRAPHS

The examples in this chapter require that you have the Big Data Library loaded. The examples are not large data sets; rather, they are small data objects that you convert to big data objects to demonstrate using the Big Data Library graphing functions.

Plotting Using Hexagonal Binning

Hexagonal binning plots are available for:

- Single plot (`plot`)
- Matrix of plots (`pairs`)
- Conditioned single or matrix plots (`xyplot`)

Functions that evaluate data over a grid in standard S-PLUS aggregate the data over the grid (such as binning the data and taking the mean in each grid cell, and then plot the aggregated values) when applied to a big data object.

Hexagonal binning is a data grouping or reduction method typically used on large data sets to clarify a spatial display structure in two dimensions. Think of it as partitioning a scatter plot into larger units to reduce dimensionality, while maintaining a measure of data clarity. Each unit of data is displayed with a hexagon and represents a bin of points in the plot. Hexagons are used instead of squares or rectangles to avoid misleading structure that occurs when edges of the rectangles line up exactly.

Plotting using hexagonal binning is the standard technique used when a plotting function that currently plots one point per row is applied to a big data object.

Plotting using hexagonal bins is available for a single plot, a matrix of plots, and conditioned single or matrix plots.

In the Census example in the section Displaying in a Simple Plot on page 98 of Chapter 4, Exploring and Manipulating Large Data Sets, demonstrates plotting using hexagonal binning. When you create a plot showing a distribution of zip codes by latitude and longitude, the following simple plot is displayed:

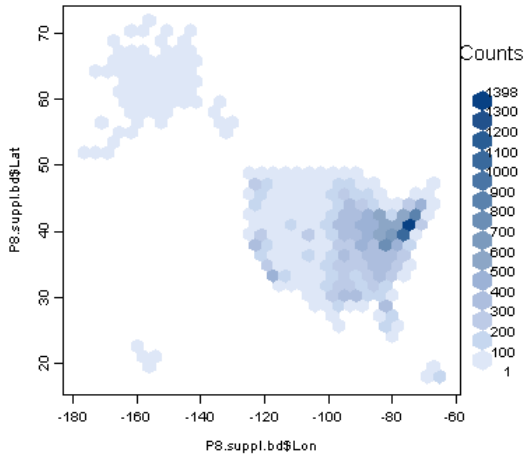


Figure 5.1: *Example of graph showing hexagonal binning.*

The functions listed in Table 5.1 support big data objects by using hexagonal binning. This section shows examples of how to call these functions for a big data object.

Create a Pair-wise Scatter Plot

The pairs function creates a figure that contains a scatter plot for each pair of variables in a `bdFrame` object.

To create a sample pair-wise scatter plot for the `fuel.frame bdFrame` object, in the **Commands** window, type the following:

```
pairs(as.bdFrame(fuel.frame))
```

The pair-wise scatter plot appears as follows:

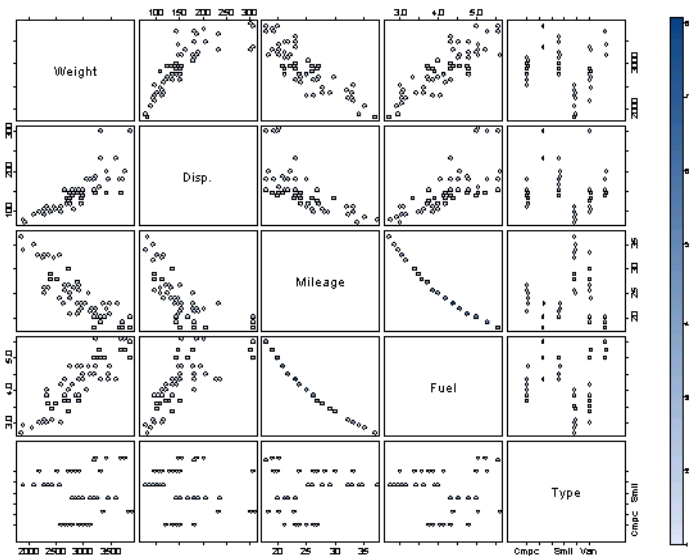


Figure 5.2: *Graph using* `pairs` *for a* `bdFrame`.

This scatter plot looks similar to the one created by calling `pairs(fuel.frame)`; however, close examination shows that the plot is composed of hexagons.

Create a Single Plot

The `plot` function can accept a `hexbin` object, a single `bdVector`, two `bdVectors`, or a `bdFrame` object. The following example plots a simple hexbin plot using the weight and mileage vectors of the `fuel.bd` object.

To create a sample single plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
plot(hexbin(fuel.bd$Weight, fuel.bd$Mileage))
```

The hexbin plot is displayed as follows:

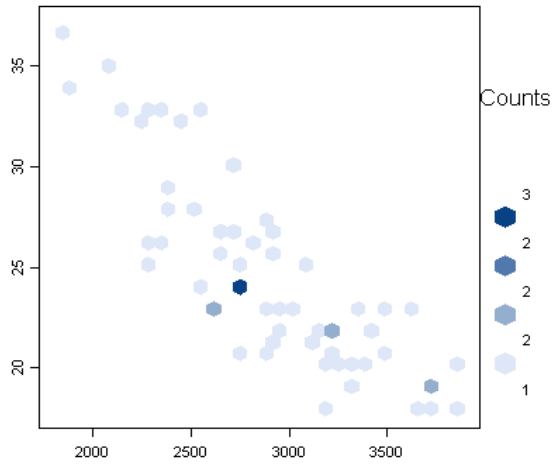


Figure 5.3: *Graph using single hexbin plot for* `fuel.bd`.

Create a Multi-Panel Scatterplot Matrix

The function `splom` creates a Trellis graph of a scatterplot matrix. The scatterplot matrix is a good tool for displaying measurements of three or more variables.

To create a sample multi-panel scatterplot matrix, where you create a hexbin plot of the columns in `fuel.bd` against each other, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
splom(~., data=fuel.bd)
```

Note

Trellis functions in the Big Data Library require the `data` argument. You cannot use formulas that refer to `bdVectors` that are not in a specified `bdFrame`.

Notice that the '.' is interpreted as all columns in the data set specified by `data`.

The `splom` plot is displayed as follows:

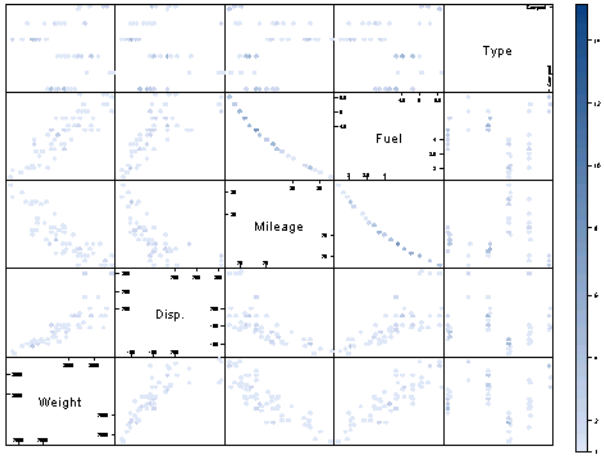


Figure 5.4: *Graph using `splom` for `fuel.bd`.*

To remove a column, use `-`*`term`*. To add a column, use `+`*`term`*. For example, the following code replaces the column `Disp.` with its log.

```
fuel.bd <- as.bdFrame(fuel.frame)
splom(~.-Disp.+log(Disp.), data=fuel.bd)
```

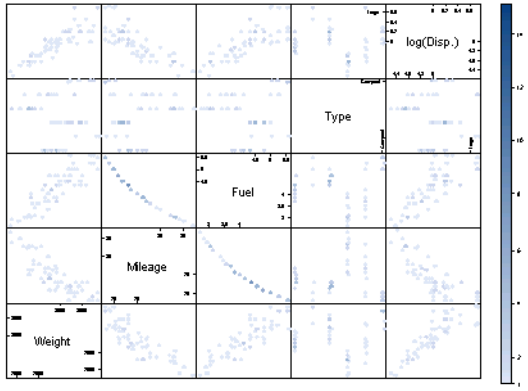


Figure 5.5: *Graph using `splom` to designate a formula for `fuel.bd`*

For more information about `splom`, see its help topic.

Create a Conditioning Plot or Scatter Plot

The function `xyplot` creates a Trellis graph, which graphs one set of numerical values on a vertical scale against another set of numerical values on a horizontal scale.

To create a sample conditioning plot, in the **Commands** window, type the following:

```
xyplot(data=as.bdFrame(air),
  ozone~radiation|temperature,
  shingle.args=list(n=4), lmline=T)
```

The variable on the left of the ~ goes on the vertical (or y) axis, and the variable on the right goes on the horizontal (or x) axis.

The function `xyplot` contains the default argument `lmline=T` to add the approximate least squares line to a panel quickly. This argument performs the same action as `panel.lmline` in standard S-PLUS.

The `xyplot` plot is displayed as follows:

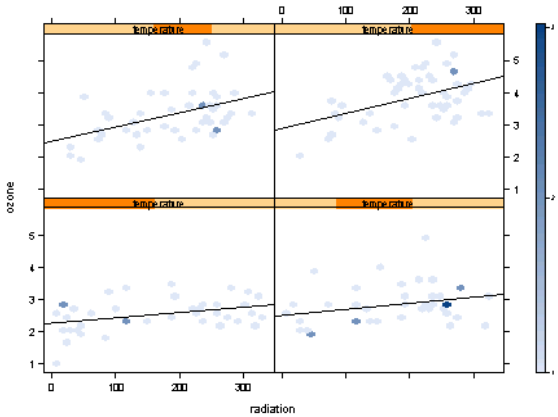


Figure 5.6: *Graph using `xyplot` with `lmline=T`.*

Trellis functions in the Big Data Library handle continuous "given" variables differently than standard data Trellis functions: they are sent through `equal.count`, rather than `factor`.

Adding Reference Lines

You can add a regression line or scatterplot smoother to hexbin plots. The regression line or smoother is a weighted fit, based on the binned values.

The following functions add the following types of reference lines to hexbin plots:

- A regression line with `abline`
- A Loess smoother with `loess.smooth`
- A smooth spline with `smooth.spline`
- A line to a `qqplot` with `qqline`
- A least squares line to an `xyplot` in a Trellis graph.

For `smooth.spline` and `loess.smooth`, when the data consists of `bdVectors`, the data is aggregated before smoothing. The range of the `x` variable is divided into 1000 bins, and then the mean for `x` and `y` is computed in each bin. A weighted smooth is then computed on the bin means, weighted based on the bin counts. This computation results in values that differ somewhat from those where the smoother is applied to the unaggregated data. The values are usually close enough to be indistinguishable when used in a plot, but the difference could be important when the smoother is used for prediction or optimization.

Add a Regression Line

When you create a scatterplot from your large data set, and you notice a linear association between the y-axis variable and the x-axis variable, you might want to display a straight line that has been fit to the data. Call `lsfit` to perform a least squares regression, and then use that regression to plot a regression line.

The following example draws an `abline` on the chart that plots `fuel.bd` weight and mileage data. First, create a hexbin object and plot it, and then add the abline to the plot.

To add a regression line to a sample plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hexbin.out <- plot(fuel.bd$Weight, fuel.bd$Mileage)
  # displays a hexbin plot
# use add.to.hexbin to keep the abline within the
# hexbin area. If you just call abline, then the
# line might draw outside of the hexbin and interfere
# with the label.
add.to.hexbin(hexbin.out, abline(lsfit(fuel.bd$Weight,
  fuel.bd$Mileage)))
```

The resulting chart is displayed as follows:

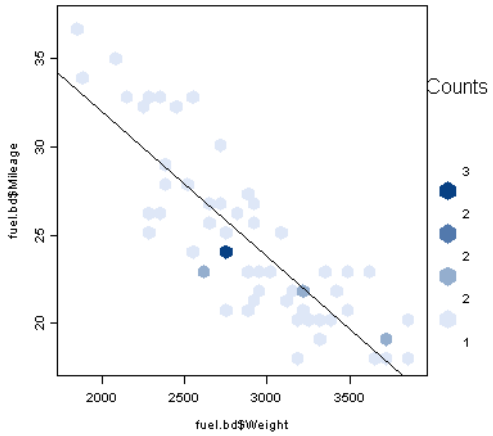


Figure 5.7: *Graph drawing an* `abline` *in a hexbin plot.*

Add a Loess Smoother

Use `lines(loess.smooth)` to add a smooth curved line to a scatter plot.

To add a `loess` smoother to a sample plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hexbin.out <- plot(fuel.bd$Weight, fuel.bd$Mileage)
  # displays a hexbin plot
add.to.hexbin(hexbin.out,
  lines(loess.smooth(fuel.bd$Weight,
  fuel.bd$Mileage), lty=2))
```

The resulting chart is displayed as follows:

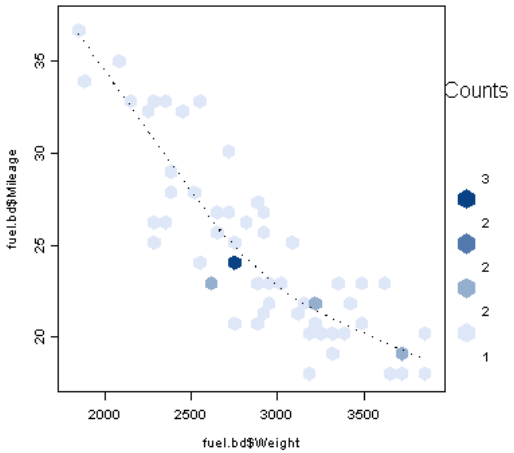


Figure 5.8: *Graph using `loess.smooth` in a hexbin plot.*

Add a Smoothing Spline

Use `lines(smooth.spline)` to add a smoothing spline to a scatter plot.

To add a smoothing spline to a sample plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hexbin.out <- plot(fuel.bd$Weight, fuel.bd$Mileage)
  # displays a hexbin plot
add.to.hexbin(hexbin.out,
  lines(smooth.spline(fuel.bd$Weight,
     fuel.bd$Mileage),lty=3))
```

The resulting chart is displayed as follows:

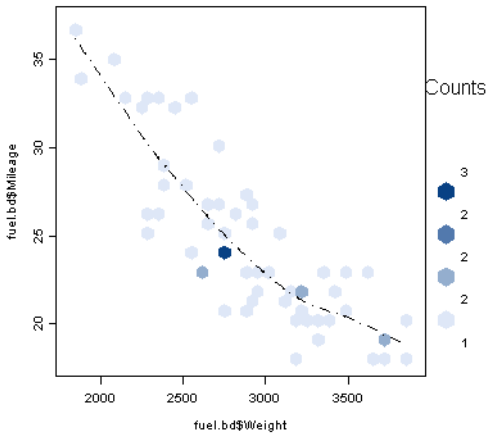


Figure 5.9: *Graph using `smooth.spline` in a hexbin plot.*

Add a Least Squares Line to an `xyplot`

To add a reference line to an `xyplot`, set `lmline=T`. Alternatively, you can call `panel.lmline` or `panel.loess`. See the section Create a Conditioning Plot or Scatter Plot on page 128 for an example.

Add a `qqplot` Reference Line

The function `qqline` fits and plots a line through a normal `qqplot`.

To add a `qqline` reference line to a sample `qqplot`, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qqnorm(fuel.bd$Mileage)
qqline(fuel.bd$Mileage)
```

The `qqline` chart is displayed as follows:

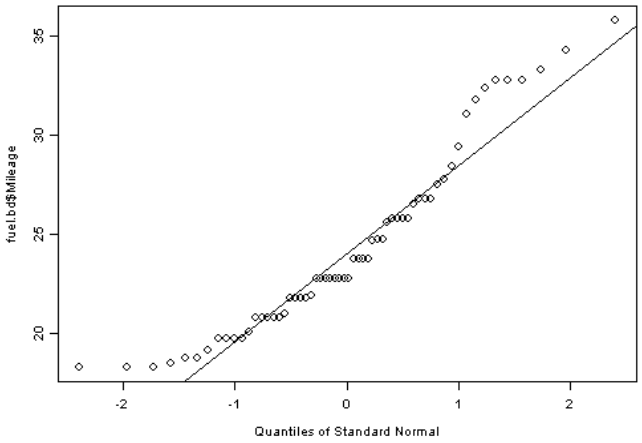


Figure 5.10: *Graph using `qqline` in a `qqplot` chart.*

Plotting by Summarizing Data

The following examples demonstrate functions that summarize data in a plot-specific manner to plot big data objects. These functions do not use hexagonal binning. Because the plots for these functions are always monotonically increasing, hexagonal binning would obscure the results. Rather, summarizing provides the appropriate information.

Create a Box Plot

The following example creates a simple box plot from `fuel.bd`. To create a Trellis box and whisker plot, see the following section.

To create a sample box plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
boxplot(split(fuel.bd$Fuel, fuel.bd$Type), style.bxp="att")
```

The box plot is displayed as follows:

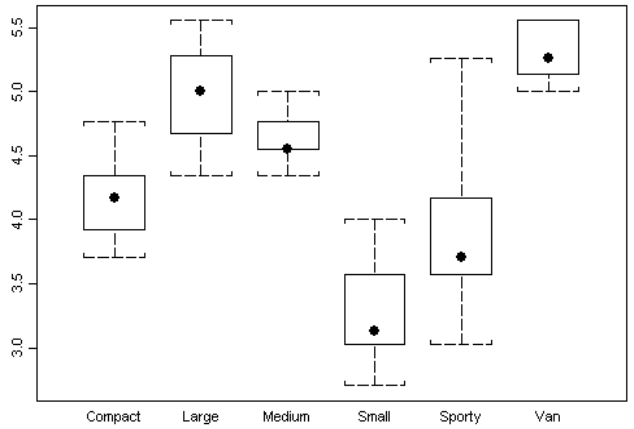


Figure 5.11: *Graph using* `boxplot`.

Create a Trellis Box and Whisker Plot

The box and whisker plot provides graphical representation showing the center and spread of a distribution.

To create a sample box and whisker plot in a Trellis graph, in the **Commands** window, type the following:

```
bwplot(Type~Fuel, data=(as.bdFrame(fuel.frame)))
```

The box and whisker plot is displayed as follows:

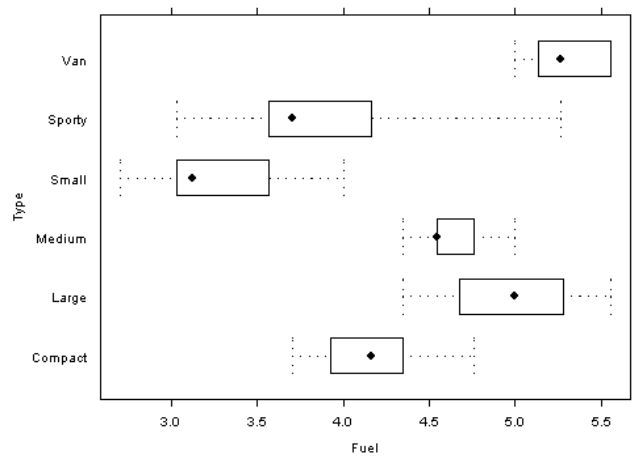


Figure 5.12: *Graph using* `bwplot`.

For more information about `bwplot`, see Chapter 3, Traditional Trellis Graphics in the *Application Developer's Guide*.

Create a Density Plot

The `density` function returns x and y coordinates of a non-parametric estimate of the probability density of the data. Options include the choice of the window to use and the number of points at which to estimate the density. Weights may also be supplied.

`Density` estimation is essentially a smoothing operation. Inevitably there is a trade-off between bias in the estimate and the estimate's variability: wide windows produce smooth estimates that may hide local features of the density.

Density summarizes data. That is, when the data is a `bdVector`, the data is aggregated before smoothing. The range of the `x` variable is divided into 1000 bins, and the mean for `x` is computed in each bin. A weighted density estimate is then computed on the bin means, weighted based on the bin counts. This calculation gives values that differ somewhat from those when `density` is applied to the unaggregated data. The values are usually close enough to be indistinguishable when used in a plot, but the difference could be important when `density` is used for prediction or optimization.

To plot density, use the `plot` function.

To create a sample density plot from `fuel.bd`, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
plot(density(fuel.bd$Weight), type="l")
```

The density plot is displayed as follows:

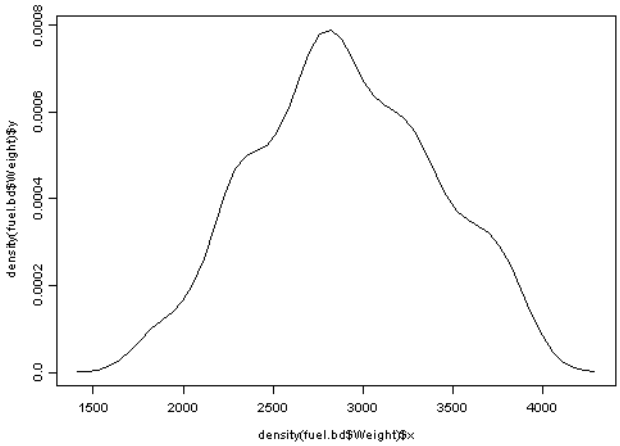


Figure 5.13: *Graph using* `density`

Create a Trellis Density Plot

The following example creates a Trellis graph of a density plot, which displays the shape of a distribution. You can use the Trellis density plot for analyzing a one-dimensional data distribution. A density plot displays an estimate of the underlying probability density function for a data set, allowing you to approximate the probability that your data fall in any interval.

To create a sample Trellis density plot, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
densityplot( ~ height | voice.part, data = singer.bd,
  layout = c(2, 4), aspect= 1, xlab = "Height (inches)",
  width = 5)
```

The Trellis density plot is displayed as follows:

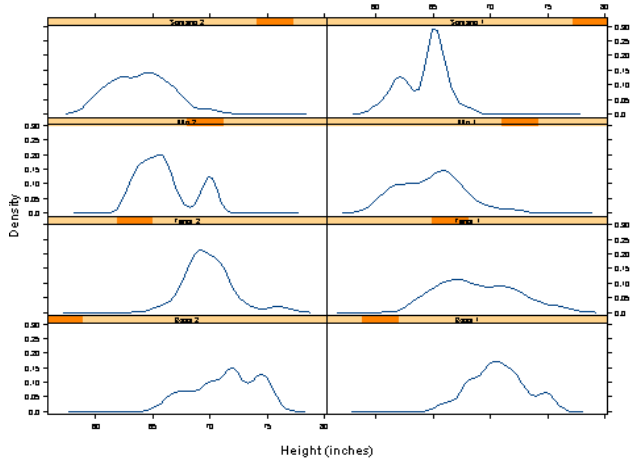


Figure 5.14: *Graph using* `densityplot`.

For more information about Trellis density plots, see Chapter 3, Traditional Trellis Graphics in the in the *Application Developer's Guide*.

Create a Simple Histogram

A histogram displays the number of data points that fall in each of a specified number of intervals. A histogram gives an indication of the relative density of the data points along the horizontal axis. For this reason, density plots are often superposed with (scaled) histograms.

To create a sample `hist` chart of a full dataset for a numeric vector, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hist(fuel.bd$Weight)
```

The numeric `hist` chart is displayed as follows:

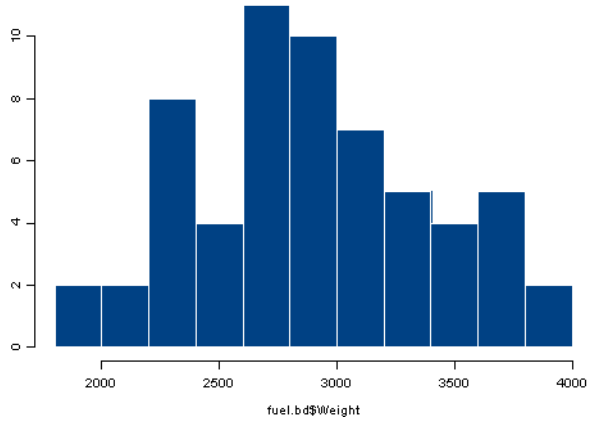


Figure 5.15: *Graph using `hist` for numeric data.*

To create a sample `hist` chart of a full dataset for a factor column, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
hist(fuel.bd$Type)
```

The factor `hist` chart is displayed as follows:

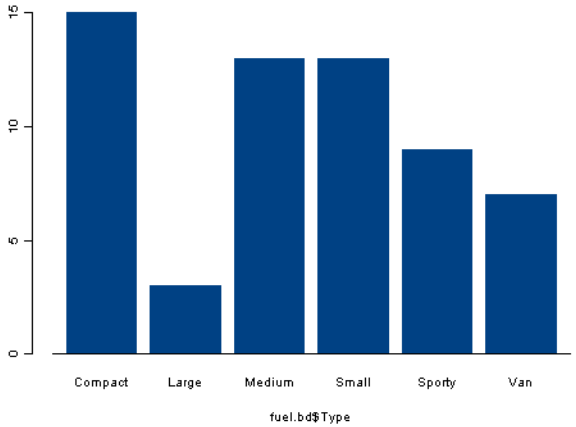


Figure 5.16: *Graph using `hist` for factor data.*

Create a Trellis Histogram

The `histogram` function for a Trellis graph is histogram.

To create a sample Trellis histogram, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
histogram( ~ height | voice.part, data = singer.bd,
  nint = 17, endpoints = c(59.5, 76.5), layout = c(2,4),
  aspect = 1, xlab = "Height (inches)")
```

The Trellis histogram chart is displayed as follows:

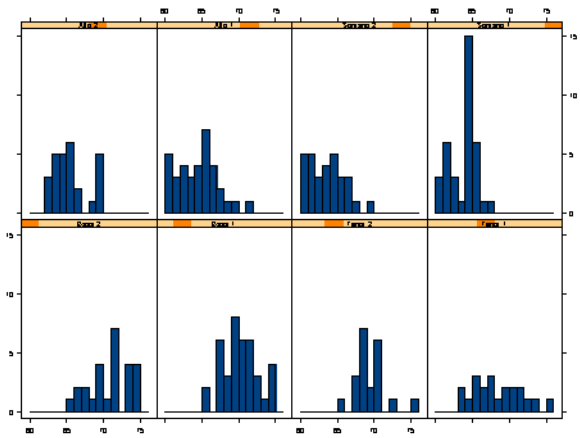


Figure 5.17: *Graph using* `histogram`.

For more information about Trellis histograms, see Chapter 3, Traditional Trellis Graphics in the in the *Application Developer's Guide*.

Create a Quantile-Quantile (QQ) Plot for Comparing Multiple Distributions

The functions `qq`, `qqmath`, `qqnorm`, and `qqplot` create an ordinary x-y plot of 500 evenly-spaced quantiles of data.

The function `qq` creates a Trellis graph comparing the distributions of two sets of data. Quantiles of one dataset are graphed against corresponding quantiles of the other data set.

To create a sample `qq` plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qq((Type=="Compact")~Mileage, data = fuel.bd)
```

The factor on the left side of the ~ must have exactly two levels (`fuel.bd$Compact` has five levels).

The `qq` plot is displayed as follows:

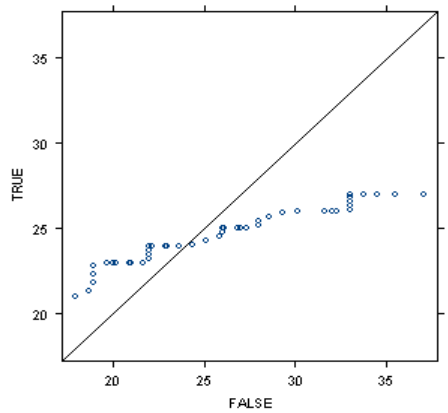


Figure 5.18: *Graph using `qq`.*

(Note that in this example, by setting `Type` to the logical `Compact`, the labels are set to `FALSE` and `TRUE` on the x and y axis, respectively.)

Create a QQ Plot Using a Theoretical or Empirical Distribution

The function `qqmath` creates normal probability plot in a Trellis graph. that is, the ordered data are graphed against quantiles of the standard normal distribution.

`qqmath` can also make probability plots for other distributions. It has an argument `distribution`, whose input is any function that computes quantiles. The default for `distribution` is `qnorm`. If you set `distribution = qexp`, the result is an exponential probability plot.

To create a sample `qqmath` plot, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
qqmath( ~ height | voice.part, data = singer.bd,
  layout = c(2, 4), aspect = 1,
  xlab = "Unit Normal Quantile",
  ylab = "Height (inches)")
```

The `qqmath` plot is displayed as follows:

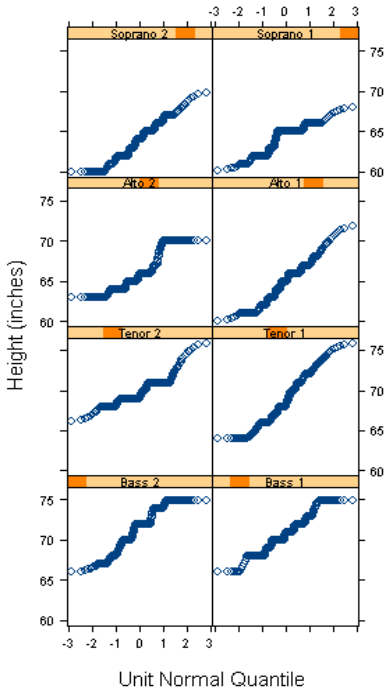


Figure 5.19: *Graph using `qqmath`.*

Create a Single Vector QQ Plot

The function `qqnorm` creates a plot using a single `bdVector` object. The following example creates a plot from the mileage vector of the `fuel.bd` object.

To create a sample `qqnorm` plot, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qqnorm(fuel.bd$Mileage)
```

The qqnorm plot is displayed as follows:

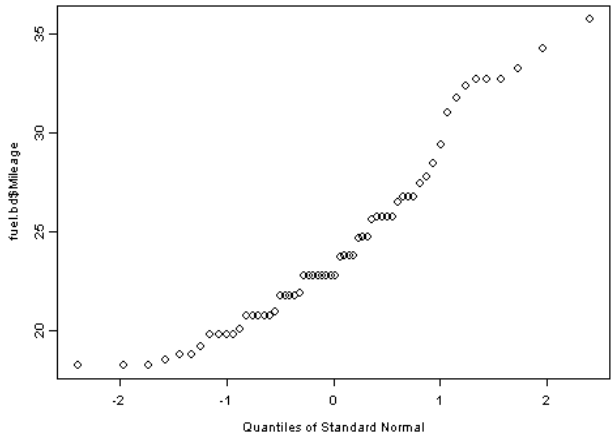


Figure 5.20: *Graph using qqnorm.*

Create a Two Vector QQ Plot

The function qqplot creates a hexbin plot using two bdVectors. The quantile-quantile plot is a good tool for determining a good approximation to a data set's distribution. In a qqplot, the ordered data are graphed against quantiles of a known theoretical distribution.

To create a sample two-vector qqplot, In the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
qqplot(fuel.bd$Mileage, runif(length(fuel.bd$Mileage),
  bigdata=T))
```

Note that in this example, the required y argument for qqplot is runif(length(fuel.bd$Mileage): the random generation for the uniform distribution for the vector fuel.bd$Mileage. Also note that using runif with a big data object requires that you set the runif argument bigdata=T.

The qqplot plot is displayed as follows:

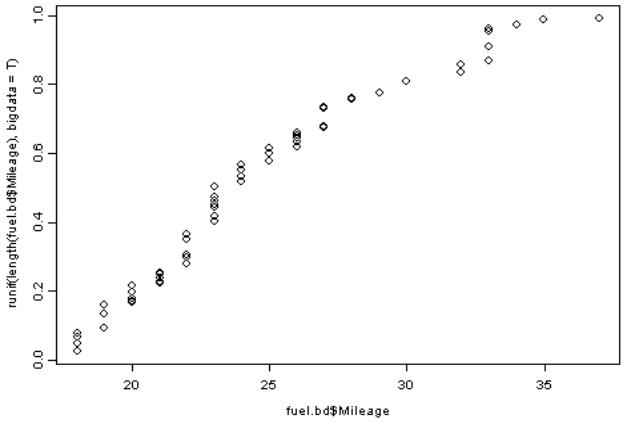


Figure 5.21: *Graph using* `qqplot`.

Create a One-Dimensional Scatter Plot

The function `stripplot` creates a Trellis graph similar to a box plot in layout; however, the individual data points are shown instead of the box plot summary.

To create sample one-dimensional scatter plot, in the **Commands** window, type the following:

```
singer.bd <- as.bdFrame(singer)
stripplot(voice.part ~ jitter(height),
   data = singer.bd, aspect = 1,
   xlab = "Height (inches)")
```

The `stripplot` plot is displayed as follows:

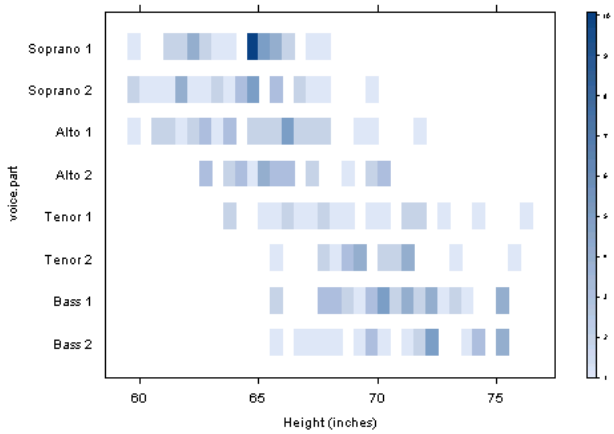


Figure 5.22: *Graph using `stripplot` for `singer.bd`.*

Creating Graphs with Preprocessing Functions

The functions discussed in this section do not accept a big data object directly to create a graph; rather, they require a preprocessing function such as those listed in the section Functions Providing Support to Preprocess Data for Graphing on page 120.

Create a Bar Chart

Calling `barchart` directly on a large data set produces a large number of bars, which results in an illegible plot.

- If your data contains a small number of cases, convert the data to a standard `data.frame` before calling `barchart`.
- If your data contains a large number of cases, first use `aggregate`, and then use `bd.coerce` to create the appropriate small data set.

In the following example, sum the yields over sites to get the total yearly yield for each variety.

To create a sample bar chart, in the **Commands** window, type the following:

```
barley.bd <- as.bdFrame(barley)
temp.df <- bd.coerce(aggregate(barley.bd$yield,
  list(year=barley.bd$year,
  variety=barley.bd$variety), sum))
barchart(variety ~ x | year, data = temp.df,
  aspect = 0.4,xlab = "Barley Yield (bushels/acre)")
```

The resulting bar chart appears as follows:

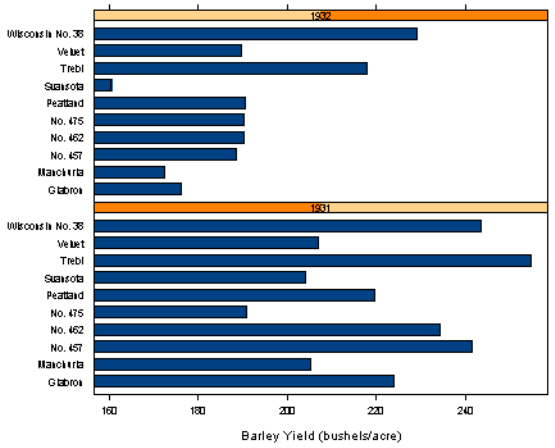


Figure 5.23: *Graph using* `barchart`.

Create a Bar Plot

The following example creates a simple bar plot from `fuel.bd`, using `table` to preprocess data.

To create a sample bar plot using `table` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
barplot(table(fuel.bd$Type), names=levels(fuel.bd$Type),
   ylab="Count")
```

The bar plot is displayed as follows:

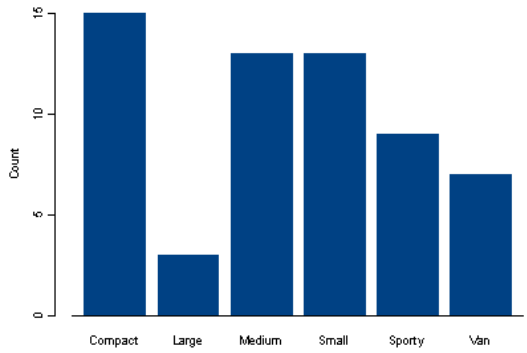


Figure 5.24: *Graph using* `barplot`.

To create a sample bar plot using `tapply` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
barplot(tapply(fuel.bd$Mileage, fuel.bd$Type, mean),
  names=levels(fuel.bd$Type), ylab="Average Mileage")
```

The bar plot is displayed as follows:

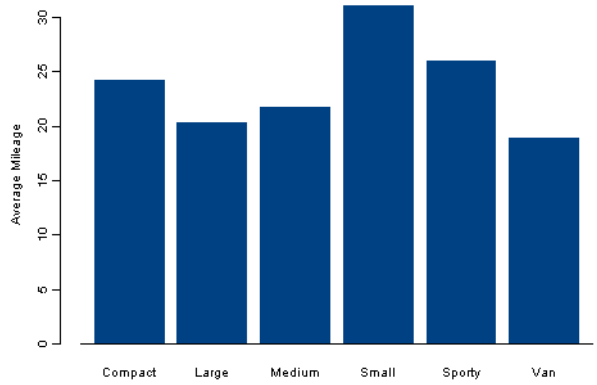


Figure 5.25: *Graph using* `tapply` *to create a bar plot.*

Create a Contour Plot

A contour plot is a representation of three-dimensional data in a flat, two-dimensional plane. Each contour line represents a height in the *z* direction from the corresponding three-dimensional surface. A level plot is essentially identical to a contour plot, but it has default options that allow you to view a particular surface differently.

The following example creates a contour plot from `fuel.bd`, using `interp` to preprocess data. For more information about `interp`, see the section Visualizing Three-Dimensional Data on page 94 of the *Application Developer's Guide*.

Like `density`, `interp` and `loess` summarize the data. That is, when the data is a `bdVector`, the data is aggregated before smoothing. The range of the `x` variable is divided into 1000 bins, and the mean for `x` computed in each bin. See the section Create a Density Plot on page 135 for more information.

To create a sample contour plot using `interp` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
contour(interp(fuel.bd$Weight, fuel.bd$Disp.,
  fuel.bd$Mileage))
```

The contour plot is displayed as follows:

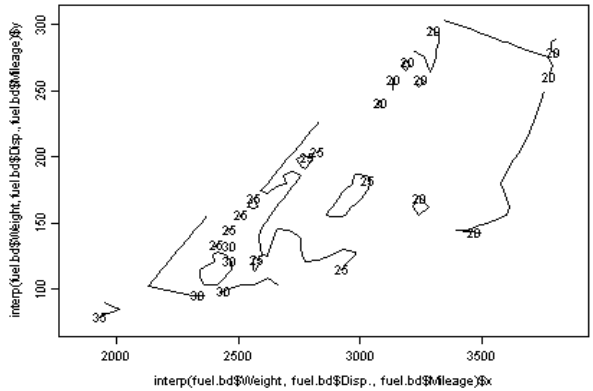


Figure 5.26: *Graph using `interp` to create a contour plot.*

Create a Trellis Contour Plot

The function `contourplot` creates a Trellis contour plot. The `contourplot` function creates a Trellis graph of a contour plot. For big data sets, `contourplot` requires a preprocessing function such as `loess`.

The following example creates a contour plot of predictions from `loess`.

To create a sample Trellis contour plot using `loess` to preprocess data, in the **Commands** window, type the following:

```
environ.bd <- as.bdFrame(environmental)
{
  ozo.m <- loess((ozone^(1/3)) ~
     wind * temperature * radiation,data = environ.bd,
     parametric = c("radiation", "wind"),
     span = 1, degree = 2)
  w.marginal <- seq(min(environ.bd$wind),
     max(environ.bd$wind), length = 50)
  t.marginal <- seq(min(environ.bd$temperature),
     max(environ.bd$temperature), length = 50)
  r.marginal <- seq(min(environ.bd$radiation),
     max(environ.bd$radiation), length = 4)
  wtr.marginal <- list(wind = w.marginal,
     temperature = t.marginal, radiation = r.marginal)
  grid <- expand.grid(wtr.marginal)
  grid[, "fit"] <- c(predict(ozo.m, grid))
  print(contourplot(fit ~ wind * temperature | radiation,
     data = grid, xlab = "Wind Speed (mph)",
     ylab = "Temperature (F)",
     main = "Cube Root Ozone (cube root ppb)"))
  }
```

The Trellis contour plot is displayed as follows:

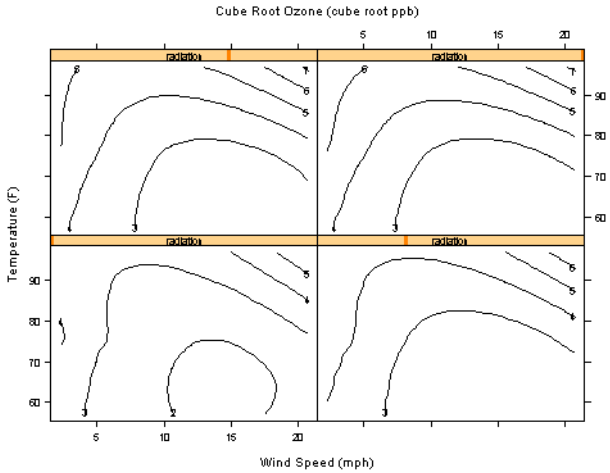


Figure 5.27: *Graph using `loess` to create a Trellis contour plot.*

Create a Dot Chart

When you create a dot chart, you can use a grouping variable and group summary, along with other options. The function `dotchart` can be preprocessed using either `table` or `tapply`.

To create a sample dot chart using `table` to preprocess data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
dotchart(table(fuel.bd$Type), labels=levels(fuel.bd$Type),
  xlab="Count")
```

The dot chart is displayed as follows:

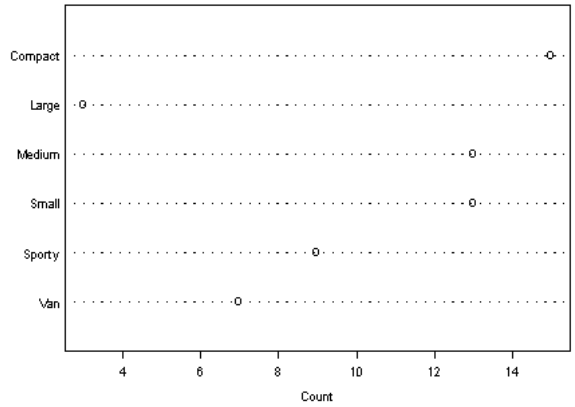


Figure 5.28: *Graph using* `table` *to create a dot chart.*

To create a sample dot chart using `tapply` to preprocess data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
dotchart(tapply(fuel.bd$Mileage, fuel.bd$Type, median),
  labels=levels(fuel.bd$Type), xlab="Median Mileage")
```

The dot chart is displayed as follows:

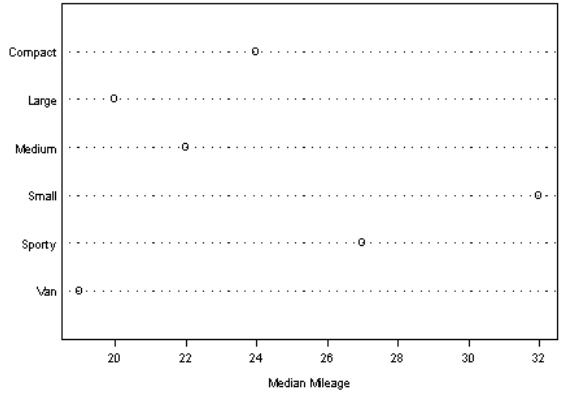


Figure 5.29: *Graph using* `tapply` *to create a dot chart.*

Create a Dot Plot

The function `dotplot` creates a Trellis graph that displays that displays dots and gridlines to mark the data values in dot plots. The dot plot reduces most data comparisons to straightforward length comparisons on a common scale.

When using `dotplot` on a big data object, call `dotplot` after using `aggregate` to reduce size of data.

In the following example, sum the barley yields over sites to get the total yearly yield for each variety.

To create a sample dot plot, in the **Commands** window, type the following:

```
barley.bd <- as.bdFrame(barley)
temp.df <- bd.coerce(aggregate(barley.bd$yield,
  list(year=barley.bd$year, variety=barley.bd$variety),
  sum))
(dotplot(variety ~ x | year, data = temp.df,
  aspect = 0.4, xlab = "Barley Yield (bushels/acre)"))
```

The resulting Trellis dot plot appears as follows:

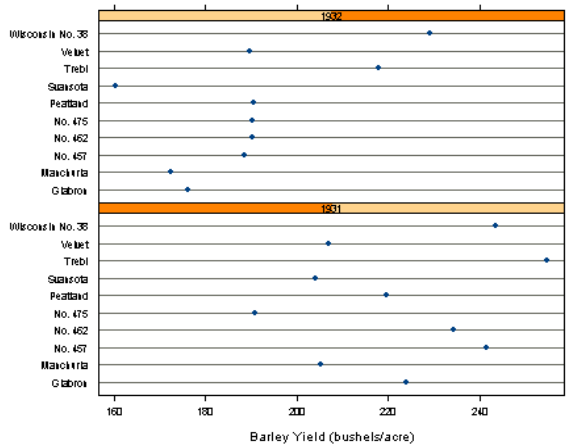


Figure 5.30: *Graph using `aggregate` to create a dot chart.*

Create an Image Graph Using `hist2d`

The following example creates an `image` graph using `hist2d` to preprocess data. The function `image` creates an image, under some graphics devices, of shades of gray or colors that represent a third dimension.

To create a sample image plot using `hist2d` preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
image(hist2d(fuel.bd$Weight, fuel.bd$Mileage, nx=9, ny=9))
```

The `image` plot is displayed as follows:

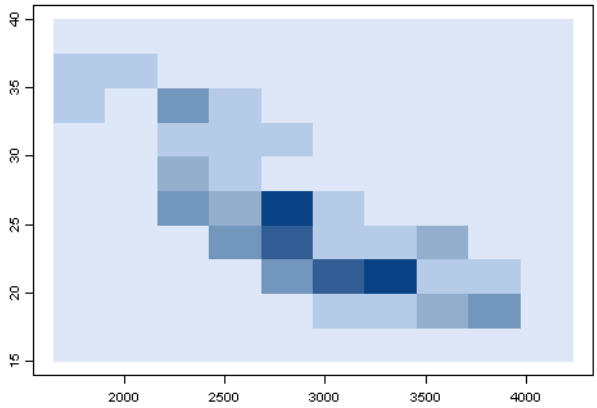


Figure 5.31: *Graph using `hist2d` to create an image plot.*

Create a Trellis Level Plot

The `levelplot` function creates a Trellis graph of a level plot. For big data sets, `levelplot` requires a preprocessing function such as `loess`.

A level plot is essentially identical to a contour plot, but it has default options so you can view a particular surface differently. Like contour plots, level plots are representations of three-dimensional data in flat, two-dimensional planes. Instead of using contour lines to indicate heights in the *z* direction, level plots use colors. The following example produces a level plot of predictions from `loess`.

To create a sample Trellis level plot using `loess` to preprocess the data, in the **Commands** window, type the following:

```
environ.bd <- as.bdFrame(environmental)
{
  ozo.m <- loess((ozone^(1/3)) ~
     wind * temperature * radiation, data = environ.bd,
     parametric = c("radiation", "wind"),
     span = 1, degree = 2)
```

```
w.marginal <- seq(min(environ.bd$wind),
    max(environ.bd$wind), length = 50)
t.marginal <- seq(min(environ.bd$temperature),
     max(environ.bd$temperature), length = 50)
r.marginal <- seq(min(environ.bd$radiation),
     max(environ.bd$radiation), length = 4)
wtr.marginal <- list(wind = w.marginal,
    temperature = t.marginal, radiation = r.marginal)
grid <- expand.grid(wtr.marginal)
grid[, "fit"] <- c(predict(ozo.m, grid))
print(levelplot(fit ~ wind * temperature | radiation,
     data = grid, xlab = "Wind Speed (mph)",
    ylab = "Temperature (F)",
    main = "Cube Root Ozone (cube root ppb)"))
}
```

The level plot is displayed as follows:

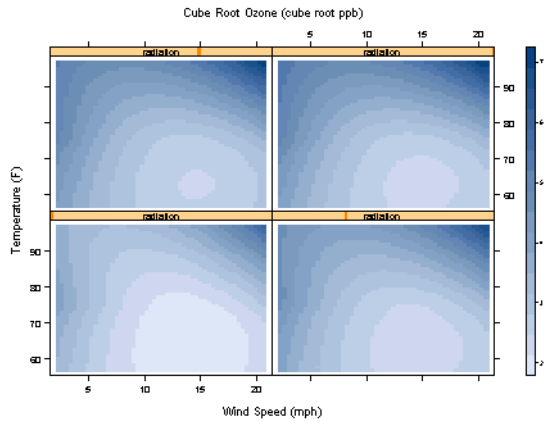


Figure 5.32: *Graph using `loess` to create a level plot.*

Create a `persp` Graph Using `hist2d`

The `persp` function creates a perspective plot given a matrix that represents heights on an evenly spaced grid. For more information about persp, see section Perspective Plots on page 96 of the *Application Developer's Guide.*

To create a sample `persp` graph using `hist2d` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
persp(hist2d(fuel.bd$Weight, fuel.bd$Mileage))
```

The `persp` graph is displayed as follows:

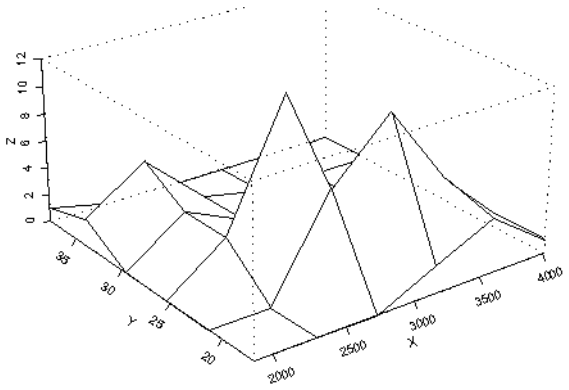


Figure 5.33: *Graph using `hist2d` to create a perspective plot*

Hint
Using `persp` of `interp` might produce a more attractive graph.

Create a Pie Chart

A pie chart shows the share of individual values in a variable, relative to the sum total of all the values. Pie charts display the same information as bar charts and dot plots, but can be more difficult to interpret. This is because the size of a pie wedge is relative to a sum, and does not directly reflect the magnitude of the data value. Because of this, pie charts are most useful when the emphasis is on an individual item's relation to the whole; in these cases, the sizes of the pie wedges are naturally interpreted as percentages.

Calling `pie` directly on a big data object can result in a pie with thousands of wedges; therefore, preprocess the data using `table` to reduce the number of wedges.

To create a sample pie chart using `table` to preprocess the data, in the **Commands** window, type the following:

```
fuel.bd <- as.bdFrame(fuel.frame)
pie(table(fuel.bd$Type), names=levels(fuel.bd$Type),
  sub="Count")
```

The pie chart appears as follows:

Figure 5.34: *Graph using `table` to create a pie chart.*

Create a Trellis Pie Chart

The function `piechart` creates a pie chart in a Trellis graph.

- If your data contains a small number of cases, convert the data to a standard `data.frame` before calling `piechart`.
- If your data contains a large number of cases, first use `aggregate`, and then use `bd.coerce` to create the appropriate small data set.

To create a sample Trellis pie chart using `aggregate` to preprocess the data, in the **Commands** window, type the following:

```
barley.bd <- as.bdFrame(barley)
temp.df <- bd.coerce(aggregate(barley.bd$yield,
  list(year=barley.bd$year, variety=barley.bd$variety),
  sum))
piechart(variety ~ x | year, data = temp.df,
  xlab = "Barley Yield (bushels/acre)")
```

The Trellis pie chart appears as follows:

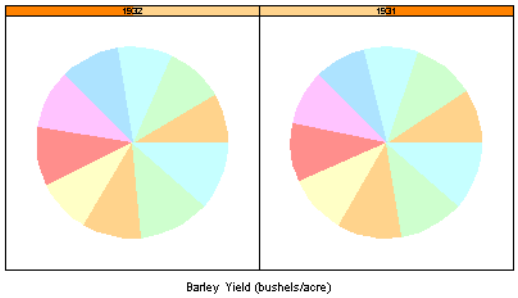


Figure 5.35: *Graph using `aggregate` to create a Trellis pie chart.*

Create a Trellis `Wireframe` Plot

A surface plot is an approximation to the shape of a three-dimensional data set. Surface plots are used to display data collected on a regularly-spaced grid; if gridded data is not available, interpolation is used to fit and plot the surface. The Trellis function that displays surface plots is `wireframe`.

For big data sets, `wireframe` requires a preprocessing function such as `loess`.

To create a sample Trellis surface plot using `loess` to preprocess the data, in the **Commands** window, type the following:

```
environ.bd <- as.bdFrame(environmental)
{
  ozo.m <- loess((ozone^(1/3)) ~
     wind * temperature * radiation, data = environ.bd,
     parametric = c("radiation", "wind"),
     span = 1, degree = 2)
  w.marginal <- seq(min(environ.bd$wind),
     max(environ.bd$wind), length = 50)
  t.marginal <- seq(min(environ.bd$temperature),
     max(environ.bd$temperature), length = 50)
  r.marginal <- seq(min(environ.bd$radiation),
     max(environ.bd$radiation), length = 4)
  wtr.marginal <- list(wind = w.marginal,
     temperature = t.marginal, radiation = r.marginal)
  grid <- expand.grid(wtr.marginal)
  grid[, "fit"] <- c(predict(ozo.m, grid))
```

```
print(wireframe(fit ~ wind * temperature | radiation,
   data = grid, xlab = "Wind Speed (mph)",
   ylab = "Temperature (F)",
   main = "Cube Root Ozone (cube root ppb)"))
}
```

The surface plot is displayed as follows:

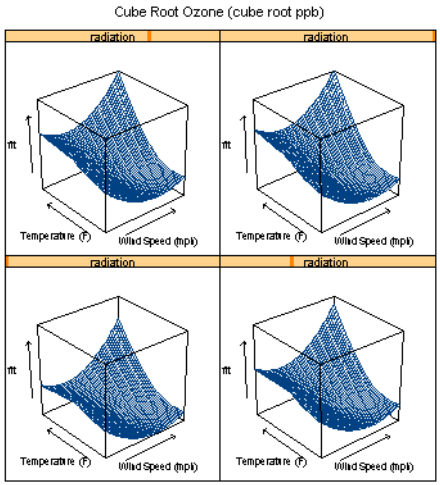


Figure 5.36: *Graph using `loess` to create a surface plot.*

Unsupported Functions

Using the functions that add to a plot, such as `points` and `lines`, results in an error message.

MODELING LARGE DATA SETS

6

Introduction **160**

Overview of Modeling **161**

Building a Model **162**
Linear Regression and Generalized Linear Modeling 162
Principal Components 169
Clustering 172

Predicting from the Model **180**
Predicting on Big Data from Small Data Models 181

INTRODUCTION

In Chapter 4, Exploring and Manipulating Large Data Sets, you graphed the filtered Census data. In this chapter, the functions available for modeling large data sets are reviewed.

In this chapter, you will perform:

- A linear regression.
- Principal components reduction.
- K-means clustering and predicting.
- Prediction from small data.

OVERVIEW OF MODELING

The Big Data library provides modeling functions on big data sets for linear models, generalized linear models (logistic regression, loglinear models, and so on), principal components and K-means clustering. In addition, you can also do prediction (scoring) with a big data sets using almost any standard S-PLUS model object that has a predict method.

The Big Data linear model, generalized linear modeling, and principal components functions are implemented using the same standard S-PLUS modeling functions: `lm`, `glm` and `princomp`, respectively. If the data argument to any of these functions is a big data object (a `bdFrame`), then S-PLUS uses the big data algorithms. Using this design, you can switch easily between working with standard and big data sets. These big data modeling functions create objects of a new class (for example, `bdLm`). Most of the standard S-PLUS methods used with modeling functions (for example, `print`, `summary`, `plot`, `predict`, `fitted`, and `residuals` work on this new class of objects.

BUILDING A MODEL

This section provides:

- An overview to linear regression, generalized linear modeling, and principal components: specifically the S-PLUS functions as they apply to large data sets.
- A list of the functions provided in the S-PLUS Big Data library for modeling.
- Exercises so you can practice modeling sample data sets.

Linear Regression and Generalized Linear Modeling

In linear regression, you model the response variable as a linear function of a set of predictor variables. Examples of response variables include sales figures and bank balances. This type of model is one of the most fundamental in nearly all applications of statistics. It has an intuitive appeal, in that it explores relationships between variables that are readily described by straight lines (or their generalizations in multiple dimensions).

If you are new to linear regression and generalized linear modeling, you might want to review their different uses:

- Use linear regression to predict a continuous response as a linear function of predictors using a least-squares fitting criterion.
- Use generalized linear modeling to predict a general response as a linear combination of the predictors using maximum likelihood.

For more information about model types, see Chapter 10, Regression and Smoothing for Continuous Response Data in *Guide to Statistics, Volume 1*.

In S-PLUS, linear regression (lm) and generalized linear modeling (glm) share many function names. Table A.12 in the Appendix, Big Data Library Functions identifies these functions as implemented for either large data linear modeling (`bdLm`), large data generalized linear modeling (`bdGlm`), or both. Implemented functions are marked with a hash mark (#) in the model type's column.

The Big Data library includes generalized linear models. Like the Big Data linear models, the Big Data generalized linear models are invoked through a call to the `glm` function when the data argument is a Big Data object (a `bdFrame`). The standard arguments to `glm`: `formula`, `family`, `data`, `subset`, `weights`, `na.action` work with Big Data. The standard model methods (`residuals`, `fitted`, `coef`, `print`, `summary`, `plot`, `anova`, `predict`) all work with Big Data `glms`.

Note
At this time the `gamma` family does not work with bigdata `glms`.

For a list of functions implemented for big data linear modeling and generalized linear modeling (and a short description of each), see Table A.12 in the Appendix, Big Data Library Functions. For more detailed information about each function, see its help file.

Fitting Data for a Linear Model

The following example uses the Boston housing data to fit a linear model. As well as fitting the linear model, the example demonstrates tasks covered in earlier chapters, including:

- importing data.
- manipulating data.
- creating simple graphs.
- adding data columns.

Boston Housing Linear Regression Example

The Boston Housing example data set is included in the example directory of your S-PLUS installation (**/samples/bigdata/boston**). The text below gives brief descriptions of each of the variables in the data set.

This data set contains the Boston house-price data of Harrison and Rubinfeld (1978) that was subsequently analyzed in Belsley et al. (1980). The table in Belsley et al. (p. 244) has various transformations already applied to the data that are not included in the bostonhousing.txt file.

The main variable of interest in the **bostonhousing.txt** data is `MEDV`, the median value of owner-occupied homes (given in the thousands of dollars). We use this as the response variable in our model and attempt to predict its values based on the other thirteen variables in the data set. For a description of the other variables, see Table 6.1.

Size

The data set is fairly small: 506 rows and 14 columns; however, to demonstrate the Big Data library modeling features, we import the data set as big data. This example would work without modification on a dataset of millions of rows.

Variables

The following table lists the bostonhousing.txt variables.

Table 6.1: *bostonhousing.txt variables.*

Variable name	Description
`AGE`	Proportion of owner-occupied units built prior to 1940.
`B`	`1000(Bk-0.63)^2`, where `Bk` is the proportion of blacks by town.
`CHAS`	Indicates whether the property bounds the Charles River (= 1 if a tract bounds the river, 0 otherwise).
`CRIM`	Per capita crime rate by town.
`DIS`	Weighted distances to five Boston employment centers.
`INDUS`	Proportion of non-retail business acres per town.
`LSTAT`	Percentage of the population that is of lower economic status.

Table 6.1: *bostonhousing.txt variables. (Continued)*

Variable name	Description
MEDV	Median value of owner-occupied homes in $1000s.
NOX	Nitric oxides concentration (parts per 10 million).
PTRATIO	Pupil-teacher ratio by town.
RAD	Index of accessibility to radial highways.
RM	Average number of rooms per dwelling.
TAX	Full-value property-tax rate per $10,000.
ZN	Proportion of residential land zoned for lots over 25,000 square feet.

Source

The data are available from the University of California Irvine Machine Learning Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html).

Note

The entire script for this example can be found in the sample directory on the S-PLUS installation CD. By default, this sample is **/samples/bigdata/boston**.

Import the data

1. In the Commands window, type:

```
boston.housing.bd <-
    importData(paste(getenv("SHOME"),
    "/samples/bigdata/boston/bostonhousing.txt",
    sep=""), stringsAsFactors=F, bigdata=T)
```

In this example, we change the default `stringsAsFactors` from `TRUE` to `FALSE`, because this example does not use levels. If you do not need to use levels, setting `stringsAsFactors` to `FALSE` can improve the speed of your data import.

2. Open the data viewer to examine the data:

```
bd.data.viewer(boston.housing.bd)
```

Summarize, manipulate, and plot the data

1. To see a summary of the data, at the Command prompt, type:

```
summary(boston.housing.bd)
```

2. To see a correlation matrix of the data, at the command prompt, type:

```
bd.cor(boston.housing.bd)
```

3. To see how the percentage of lower economic status relates to housing value, create a scatterplot:

```
plot(boston.housing.bd$LSTAT, boston.housing.bd$MEDV)
```

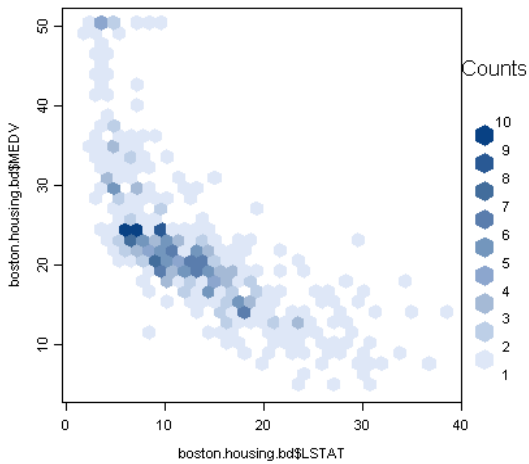


Figure 6.1: *Plot showing economic status to housing value.*

4. Compute the logarithm of MEDV and add it to boston.housing.bd object:

```
boston.housing.bd$LMEDV <-
    log(boston.housing.bd$MEDV)
```

   This requires two passes over the data: one to compute the log and one to add the new variable, called `LMEDV`, to the original data object. A more efficient method is to use the `bd.create.columns` function:

```
boston.housing.bd <- bd.create.columns(
    boston.housing.bd, exprs = "log(MEDV)",
    names = "LMEDV")
```

5. To see the relationship between distance to employment centers and the logarithm calculated, use the `plot` command:

```
plot(boston.housing.bd$DIS,
    boston.housing.bd$LMEDV)
```

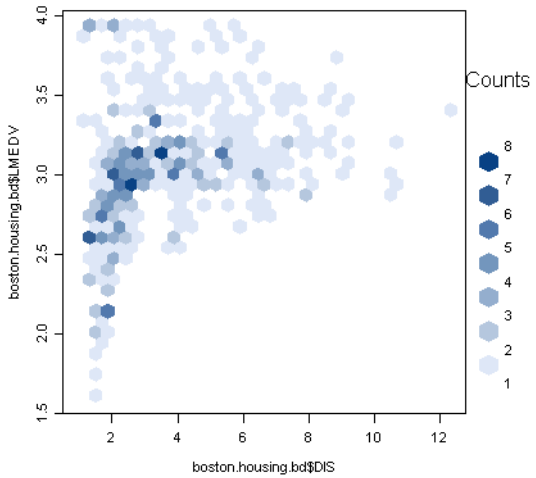


Figure 6.2: *Plot of distance to employment centers.*

6. Based on scatterplots of log housing values versus the other predictors (not shown here), we decide to account for the nonlinear relationships by transforming five of the predictor variables. Use `bd.create.columns` to create all the new variables in one pass through the data.

```
boston.housing.bd <-
    bd.create.columns(boston.housing.bd,
    exprs = c("log(RAD)", "log(LSTAT)", "NOX^2",
    "log(DIS)", "RM^2"),
    names = c("LRAD", "LLSTAT", "NOX2", "LDIS", "RM2"))
```

7. Open the data viewer and examine the new columns.

```
bd.data.viewer(boston.housing.bd)
```

8. Fit the linear regression.

```
boston.lm <-lm (LMEDV ~ CRIM + ZN + INDUS + CHAS
    + AGE + TAX + PTRATIO + B + LRAD + LLSTAT + NOX2
    + LDIS + RM2, data = boston.housing.bd)
```

9. Look at the model results by typing in the **Commands** window:

```
boston.lm
```

10. Look at some diagnostic plots for the model:

```
plot(boston.lm)
```

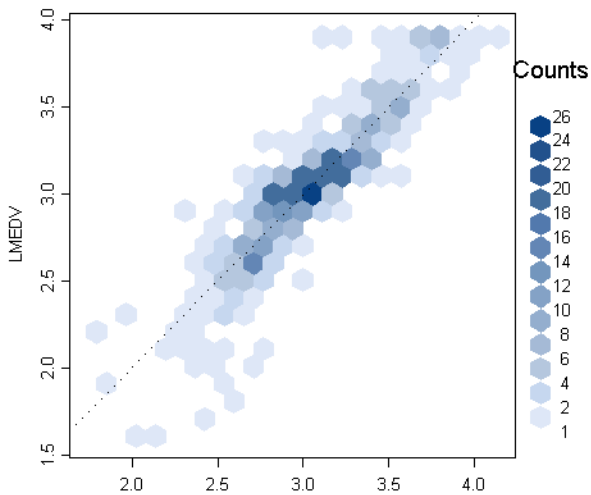


Figure 6.3: *One diagnostic plot.*

11. Call summary for a longer synopsis of the model.

```
summary(boston.lm)

Call: bdLm(formula =
LMEDV ~ CRIM + ZN + INDUS + CHAS +
AGE + TAX + PTRATIO + B + LRAD +
LLSTAT + NOX2 + LDIS + RM2,

Residuals:
 Min.  Mean  Max.  StDev
 -0.7118 0.0000 0.7978 0.1801


Coefficients:

        Value Std. Error t value Pr(>|t|)
(Intercept)  4.5578  0.1544  29.5116  0.0000
    CRIM -0.0119  0.0012  -9.5320  0.0000
     ZN  0.0001  0.0005   0.1585  0.8741
   INDUS  0.0002  0.0024   0.1013  0.9193
    CHAS  0.0914  0.0332   2.7527  0.0061
    AGE  0.0001  0.0005   0.1724  0.8632
    TAX -0.0004  0.0001  -3.4261  0.0007
  PTRATIO -0.0311  0.0050  -6.2081  0.0000
     B  0.0004  0.0001   3.5271  0.0005
    LRAD  0.0957  0.0191   5.0021  0.0000
   LLSTAT -0.3712  0.0250  -14.8406  0.0000
    NOX2 -0.6380  0.1131  -5.6393  0.0000
    LDIS -0.1913  0.0334  -5.7275  0.0000
    RM2  0.0063  0.0013   4.8226  0.0000
```

Notice that `ZN`, `INDUS`, and `AGE` are not significant predictors. If we were building a model for this data, we would likely refit several other candidate models and examine them more fully.

Principal Components

For investigation involving a large number of observed variables, it is often useful to simplify the analysis by considering a smaller number of linear combinations of the original variables. PCA is one method for this data reduction. It finds linear combinations of the data that are orthogonal and, taken together, explain all of the variance of the original data. the linear combinations from PCA can be ordered based on the variability in the original data that each one explains. It might be possible, due to redundancy in the variables, to reduce the dimension of the data by using PCA, yet still retain most of the original variability in the data.

Using principal components, you can reduce the number of predictor variables and compute values to use as predictors in a logistic regression. Take care when using the principal components as predictors for a response variable, because the principal components are computed independently of the response variable. Retention of the principal components that have the highest variance is not the same as choosing those principal components that have a highest correlation with the dependent variable.

Note
The signs of the loadings might differ between `princomp` and `bdPrincomp`, because the signs are not uniquely determined.

The Big Data library provides the **Principal Component** functions listed below.

For more detailed information on each function, see its help topic.

Table 6.2: *Principal components functions.*

Function name	Description
`loadings`	Returns the loadings component of an object.
`predict`	Computes principal component variables for new observations.
`print`	Prints the input.
`screeplot` or `plot`	Produces a barplot of the variances of the derived variables.
`summary`	Provides a summary of principal components.

Prim4 Principal Components Example

This example uses the data set provided with S-PLUS, Prim4. Prim4 is a relatively small data set (500 rows and 4 columns), but for demonstration purposes, convert it to a big data object.

1. Convert Prim4 to a big data object.

```
prim4.bd <- as.bdFrame(prim4)
```

2. Create a `primcomp` object from `prim4.bd`. `primcomp` returns an object of class `bdPrincomp`, containing the standard deviations of the principal components, the loadings, and, optionally, the scores.

```
prim4.bdp <- princomp(prim4.bd)
```

3. Get the loadings for `prim4.bdp`.

```
loadings(prim4.bdp)
```

4. Produce a plot.

```
plot(prim4.bdp)
```

The plot displays as follows:

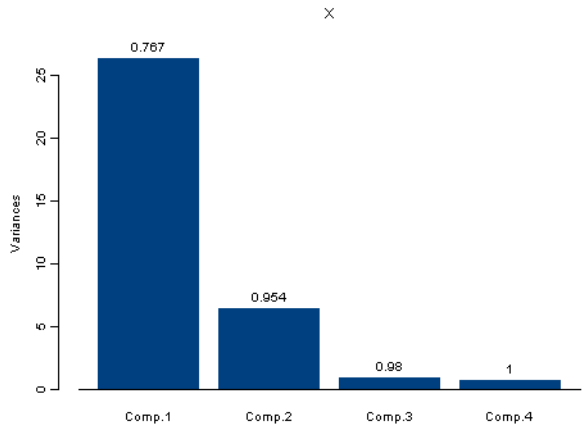


Figure 6.4: *prim4.bdp*

5. Call `predict` to extract the fitted values.

```
predict(prim4.bdp)
```

```
**bdFrame: 500 rows, 4 columns**
      Comp.1   Comp.2    Comp.3    Comp.4
1 9.6113930 1.257928 0.48919465 0.87537112
2 -4.8931668 -3.164171 -0.29226528 -0.68005429
```

```
3 -4.9597341 -2.940688 -0.23079213 -0.66704590
4 0.8345442 -1.726552 -0.09256986 -0.10535579
5 -6.6856195 2.087905 0.42910847 0.08836129
  ... 495 more rows ...
```

Note

To increase the number of rows of output data displayed, increase the `print.bdFrame.rows` value using `bd.options` (for example, `bd.options(print.bdFrame.rows=15)`).

6. To display the standard deviations and observation information, just print the object:

```
print(prim4.bdp)

bdPrincomp(x = prim4.bd)

Standard deviations:
  Comp.1  Comp.2  Comp.3  Comp.4
 5.133588 2.533057 0.9316154 0.8374292

The number of variables is 4 and the number of
observations is 500
```

7. To get more details on the components, use the `summary` function:

```
summary(prim4.bdp)

Importance of components:

                       Comp.1     Comp.2    Comp.3     Comp.4
Standard deviation 5.1335885 2.5330575 0.93161540 0.8374292

Proportion of Variance 0.7674509 0.1868524 0.02527446
0.0204223

Cumulative Proportion 0.7674509 0.9543032 0.97957770
1.0000000
```

Clustering

Cluster analysis segments observations into classes, or clusters, so that the degree of similarity is strong between members of the same cluster and weak between members of different clusters.

If you are involved in market research, you could use clustering to group respondents according to their buying preferences. If you are performing medical research, you may be able to better determine treatment if diseases are properly grouped.

Purchases, economic background, and spending habits are just a few examples of information that can be grouped, and once these objects are grouped, you can then apply this knowledge to reveal patterns and relationships on a large scale.

K-means is one of the most widespread clustering methods. It was originally developed for situations in which all variables are continuous, and the Euclidian distance is chosen as the measure of dissimilarity. There are several variants of the K-means clustering algorithm, but most variants involve an iterative scheme that operates over a fixed number of clusters while attempting to satisfy the following properties:

- Each class has a center which is the mean position of all the samples in that class.
- Each object is in the class whose center it is closest to.

The Big Data library clustering function, `bdCluster`, applies a K-means algorithm that performs a single scan of a data set, while using a buffer for points from the data set of fixed size.

Categorical data is handled by expanding categorical columns into m indicator columns, where m is the number of unique categories in the column. The K-means algorithm selects k of the objects, each of which initially represents a cluster mean or centroid. For each of the remaining objects, an object is assigned to the cluster it resembles the most, based on the distance of the object from the cluster mean. It then computes the new mean for each cluster. This process iterates until the function converges. A second scan through the data assigns each observation to the cluster it is closest to, where closeness is measured by the Euclidean distance.

When you perform K-means clustering, the number of cluster iterations you specify determines the accuracy of each cluster. That is, the higher the iteration number, the more accurate the observations.

The clustering function `bdCluster` includes the optional arguments listed in Table 6.3 for using the K-means algorithm:

Table 6.3: *bdCluster algorithm arguments.*

Optional argument	Description
`columns`	The names of columns to use in clustering. The default uses all columns.
`iter.max`	The maximum number of iterations to run within a block. This is the number of iterations of the standard K-Means algorithm applied to the combined new data from the block, the retained set, and the current centers.
`k`	The number of clusters. You might know this number based on the subject matter. For example, you know in advance you expect to find three species groups in a particular dataset. Often, however, clustering is an exploratory technique, and the number of clusters is unknown. Try a number of cluster runs with varying number of clusters and see which setting provides meaningful results.
`retain`	The number of rows in the retained set. As each block of data is processed, observations that do not cluster well are kept in the retain set. At the next step in the algorithm, the observations are added to the new chunk of data and the K-means clustering is run on this combined set.
`start`	The method for selecting starting values for centers. • Specify `"firstSample"` to use a random sample of K rows from the first block of data as the initial centers. • Specify `"kPoints"` to use the first unique K rows of data as the initial centers. • Specify `"hClustFirstBlock"` to compute the initial centers from the first block of dataset using the hierarchical clustering method. • Specify `"entireSample"` to compute the initial centers from a sample of the entire dataset using the hierarchical clustering method.

Census Clustering Example

In this section, practice performing clustering on the census data example that you filtered and graphed in the previous chapters.

> **Note**
>
> This exercise picks up from the manipulated Census data set from the end of Chapter 4. If you are starting this example at this point, without having worked through the previous chapter's exercises, you can load and run the previous exercise steps of the example script from the S-PLUS sample directory, by default installed at your installation directory in **/samples/bigdata/census**.

To perform K-means cluster analysis

1. Set the number of clusters to solve for. In this case, we set the cluster number to 40. When you model your own large data set, you can set it to a higher or lower number, depending on the data set size and the degree of accuracy you want for the clusters.

   ```
   NK <- 40
   ```

2. Set the random number generator seed for reproducibility.

   ```
   set.seed(21)
   ```

3. Call `bdCluster`, passing your normalized large data set as the data argument. Provide column names and the cluster number. Assign the resulting object to `cluster.bd`.

   ```
   cluster.bd <- bdCluster(P8.Nz.bd,
      columns=column.names.Nz, k=NK )
   ```

4. Extract the predicted cluster groups from the cluster object with the `predict` function, and then calling `cbind` to bind the resulting prediction to your normalized data set with `cbind`. Assign the resulting object to `cluster.p.bd`.

   ```
   cluster.p.bd <- cbind(P8.Nz.bd, predict(cluster.bd))
   ```

5. Display the resulting data in the data viewer. Your data set should contain 32,165 rows.

   ```
   bd.data.viewer(cluster.p.bd)
   ```

Analyze and Graph the Resulting Clusters

In the next section, analyze the clusters that you created above.

During this exercise, produce a series of histograms that illustrate each clusters' age distributions by gender. (At this point, you use no geographical information.)

In this example, you will produce two different summaries of the clusters:

- The mean histogram within each cluster group
- The number (count) of members of each cluster group.

Aggregate and order the cluster group

1. Code the cluster ID into the variable `PREDICT.membership`.

```
cluster.pm.bd <- bd.aggregate( x=cluster.p.bd,
   by.columns="PREDICT.membership",
   input.columns=column.names.Nz, summary.fns="mean")

cluster.pc.bd <- bd.aggregate( x=cluster.p.bd,
   by.columns="PREDICT.membership", input.columns=1,
   summary.fns="count")
```

2. Optionally, you can display the changed data in the data viewer.

```
bd.data.viewer(cluster.pc.bd)
```

3. Assign the mean cluster group to `cluster.pm.df` as a `bdFrame`.

```
cluster.pm.df <- bd.coerce(cluster.pm.bd)
```

4. Assign the count cluster group to `cluster.pc.df` as a `bdFrame`.

```
cluster.pc.df <- bd.coerce(cluster.pc.bd)
```

5. Assign both cluster group `bdFrames` to `cluster.pmc.df`.

```
cluster.pmc.df <- merge(cluster.pm.df, cluster.pc.df)
```

6. For a more systematic display, re-order by number of members within each cluster.

```
cluster.pmc.df <- cluster.pmc.df[rev
   (order(cluster.pmc.df$ZCTA5.count)),]
```

7. Assign the cluster ID column of the data frame (`PREDICT.membership`) to the `bdCharacter` object `PREDICT.membership.ordered`.

```
PREDICT.membership.ordered <-
   as.character(cluster.pmc.df$PREDICT.membership)
```

To prepare the graph display

1. Set the color for the histograms to cycle through the 16-color list.

   ```
   index16 = 1+((0:200)%%16)
   ```

2. Prepare the graph display. The function `graph.setup` is defined in the included file **graph.setup.q**. The function `my.vbar` is defined in the included file **my.vbar.q**. (See the section Loading Supporting Source Files on page 94 for more information.)

   This code uses the appropriate display device for both the Windows and Unix platforms.

   ```
   graph.setup(Name="Histograms")
   par(mfrow=c(5,10))
   Nplot<-30
   for(k in 1:Nplot)

   {
      my.vbar(cluster.pmc.df, k=k, plotcols=2:37,
       Nreport.col=38, col=1+index16[k] )
   }
   ```

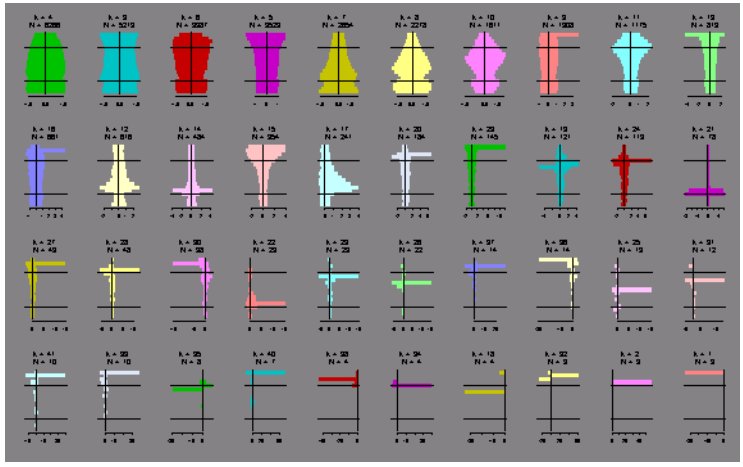


Figure 6.5: *Histograms displaying clusters.*

3. Select the columns to determine the data you want to appear in the histogram and assign them to the data frame `cluster.psbu.df`.

```
cluster.psub.df <- bd.coerce(
    bd.select.rows(x=cluster.p.bd,
    columns=c("Lat","Lon","PREDICT.membership")) )
```

4. Optionally, you can view this three-column data set in the data viewer. Observe that it still has 32,165 rows.

```
bd.data.viewer(cluster.psub.df)
```

5. Create a vector to contain the data set's latitudes.

```
Lat.vec <- cluster.psub.df$Lat
```

6. Create a vector to contain the data set's longitudes.

```
Lon.vec <- cluster.psub.df$Lon
```

7. Create a character vector to contain the data set's predicted membership.

```
Memb.vec <-
    as.character(cluster.psub.df$PREDICT.membership )
```

8. Create a vector of the column `PREDICT.membership`.

```
Memb.vec <- cluster.p.bd$PREDICT.membership
```

Creating a Multi-tabbed Sheet

In the following exercise, use the data you sorted and filtered in the previous exercise to create a multi-tabbed sheet, one for each of the first 20 clusters of your 40-cluster set.

Each sheet shows black dots for all but that sheet's salient cluster, which is superimposed with the color assigned for that sheet.

To create the multi-tabbed histogram sheet

1. Set the vector to `1:NK`, where `NK` is the cluster number.

```
Kvec=1:NK
```

2. Set up and name the histogram.

```
graph.setup(Name="USA")
```

3. Plot 20 clusters, one per tab, to create maps displaying age and gender population distribution for each cluster. Note that the histogram legend, showing age and gender distribution, appears on each tab.

```
par(err=-1)
for(k in 1:20)
{
    k.index = Memb.vec==PREDICT.membership.ordered[k]
        par(plt=c(.1,1,.1,1))
    plot(Lon.vec,Lat.vec,pch=1, cex=0.3,
        col=1,xlim=c(-125,-70), ylim=c(25,50),
        xlab="Lon",ylab="Lat")
    points(Lon.vec[k.index], Lat.vec[k.index],
        col=1+index16[k], cex=0.4, pch=16)

par(new=T)
par(plt=c(.1,.3,.1,.3))
my.vbar(cluster.pmc.df, k=k, plotcols=2:37,
    Nreport.col=38, col=1+index16[k] )
box()
}
```

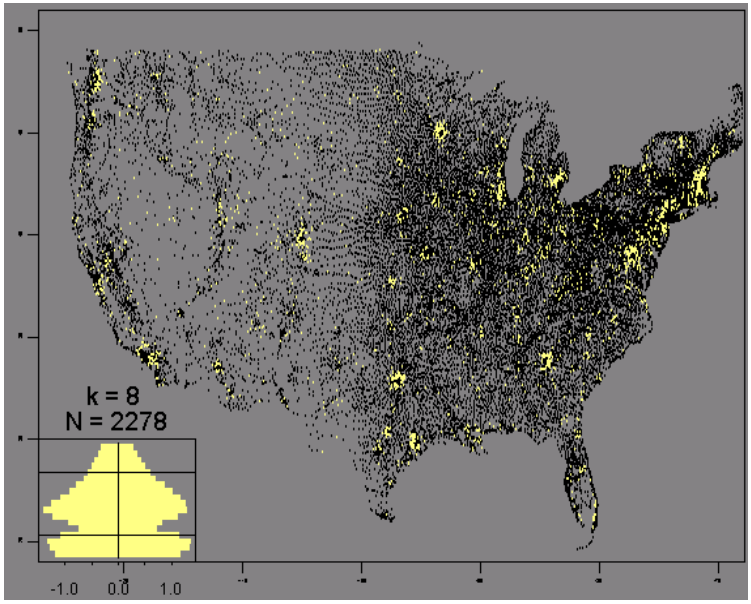


Figure 6.6: *Sample population distribution histogram.*

PREDICTING FROM THE MODEL

Other books in the S-PLUS documentation discuss at length predicting from a model, including predicting from a linear model, a generalized linear model, a generalized additive model, principal components, and clustering. For more information about predicting, see the S-PLUS Guide to Statistics, Volume 1.

The S-PLUS Big Data library provides support for predicting for most model types using big data as the data to predict for. The predict functions include the following.

Table 6.4: *Big Data library predict functions.*

Function	Predicts for this model object
`predict.bs`	Basis matrix for polynomial splines
`predict.censorReg`	Regression model for censored data
`predict.discrim`	Normal (Gaussian) linear or quadratic discriminant function
`predict.factanal`	Factor analysis model (`factanal` object)
`predict.gam`	Generalized additive model
`predict.gls`	Generalized least squares model
`predict.gnls`	Nonlinear model using generalized least squares
`predict.lm`	Linear model
`predict.lme`	Linear mixed-effects models
`predict.lmList`	List of linear model objects
`predict.lmRobMM`	Robust fit of a linear regression model, as estimated by the `lmRobMM` function.

Table 6.4: *Big Data library predict functions. (Continued)*

Function	Predicts for this model object
`predict.loess`	Local regression model
`predict.mlm`	Multiple response linear least squares model
`predict.nlme`	Nonlinear mixed-effects model
`predict.nls`	Nonlinear regression model via least squares
`predict.ns`	Basis matrix for natural splines.
`predict.princomp`	Principal components
`predict.survreg`	Parametric survival regression model
`predict.survReg`	Survival model using parametric regression.

Predicting on Big Data from Small Data Models

Many of the modern modeling methods in S-PLUS do not work on big data objects in version 7. Often, the algorithms for these models require all the data to be in memory at once.

An approach to using these in-memory models is to sample from the large data set and fit the model to the in-memory sample. The fitted model can then be used to predict all observations since the predict methods will work on bigdata objects.

In this exercise, we sample from the boston housing data (even though it is small), fit a tree model to predict the median housing value, and then use that model to predict housing median housing values for all observations in the data set. While the boston data does not require out-of-memory model fitting, for the purpose of this example we set the `max.block.size` to `100` to process the data in blocks.

Fitting the model

To fit the model:

1. If you have not done so, create the `boston.housing bdFrame` by importing the data from the samples directory:

```
boston.housing <- importData(paste(getenv("SHOME"),
   "/samples/bigdata/boston/bostonhousing.txt",
    sep=""), stringsAsFactors=F, bigdata=T)
```

2. Set the `max.block.size` to 100 to process the data in blocks:

```
bd.options(max.block.size=100)
```

3. Create a random sample of size 200 from the big data object and convert it to a data frame:

```
boston.housing.sample <-
    bd.coerce(bd.sample(boston.housing, n=200))
```

4. Fit a tree model to predict median housing value using all the rest of the variables as predictors, and then examine its summary:

```
tree.boston <-
   tree(MEDV ~ ., data=boston.housing.sample)
summary(tree.boston)
```

5. Use the tree model to predict median housing values for all observations in the boston housing data set. Plot the observed versus predicted housing values. The plot is drawn as a hexbin plot, because the predicted values (as well as the observed values) are big data objects:

```
predict.boston <-
   predict(tree.boston, boston.housing)
plot(predict.boston, boston.housing$MEDV)
```

   This model could be applied to a data model that included millions of points.

About Tree Models

A tree model is one example of models that cannot be fit on bigdata objects, but the resulting model can be used to predict all observations.

In the above example, we just made a single call to the tree function to create our tree model object. In a real modeling situation, you would likely consider several different tree models and use some of the associated tree functions, such as `cv.tree`, `prune.tree` and

`plot.tree` to select an appropriate model. For more information about tree models, see Chapter 19, Classification and Regression Trees in the *Guide to Statistics, Volume 2.*

In the above example, we did not transform any of the predictor variables for the tree model, as we did when fitting the linear regression model to the same data earlier in this chapter. The transformations are not necessary because tree models are invariant to monotone re-expression of the predictor variables. This property, along with the ease of interpretation of the resulting tree are some of the reasons why tree models are popular.

However, a disadvantage of tree models is their sensitivity; if you repeat the above sample / fit exercise, you will most likely get quite different trees. One way to overcome this problem is to aggregate multiple trees by averaging predictions from many different trees. See the literature on bagging and boosting of trees. Hastie et al. (2001) has a good overview of this technique.

References

Belsley, D., Kuh, E. and Welsch, R. (1980). *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity.* John Wiley & Sons: New York.

Harrison, D. and Rubinfeld, D. L. (1978). Hedonic prices and the demand for clean air. *Journal of Environmental Economics & Management* 5:81-102.

ADVANCED PROGRAMMING INFORMATION

7

Introduction **186**

Big Data Block Size Issues **187**
Block Size Options 187
Group or Window Blocks 188

Big Data String and Factor Issues **191**
String Column Widths 191
String Widths and importData 191
String Widths and `bd.create.columns` 193
Factor Column Levels 194
String Truncation and Level Overflow Errors 195

Storing and Retrieving Large S Objects **197**
Managing Large Amounts of Data 197

Increasing Efficiency **199**
`bd.select.rows` 199
`bd.filter.rows` 199
`bd.create.columns` 200

INTRODUCTION

As an S-PLUS Big Data library user, you might encounter unexpected or unusual behavior when you manipulate blocks of data or work with strings and factors.

This section includes warnings and advice about such behavior, and provides examples and further information for handling these unusual situations.

Alternatively, you might need to implement your own big-data algorithms using out-of-memory techniques.

BIG DATA BLOCK SIZE ISSUES

Big data objects represent very large amounts of data by storing the data in external files. When a big data object is processed, pieces of this data are read into memory and processed as data "blocks." For most operations, this happens automatically. This section describes situations where you might need to understand the processing of individual blocks.

Block Size Options

When processing big data, the system must decide how much data to read and process in each block. Each block should be as big as possible, because it is more efficient to process a few large blocks, rather than many small blocks. However, the available memory limits the block size. If space is allocated for a block that is larger than the physical memory on the computer, either it uses virtual memory to store the block (which slows all operations), or the memory allocation operation fails.

The size of the blocks used is controlled by two options:

- `bd.options("block.size")`
  The option `"block.size"` specifies the maximum number of rows to be processed at a time, when executing big data operations. The default value is `1e9`; however, the actual number of rows processed is determined by this value, adjusted downwards to fit within the value specified by the option `"max.block.mb"`.
- `bd.options("max.block.mb")`
  The option `"max.block.mb"` places a limit on the maximum size of the block in megabytes. The default value is 10.

When S-PLUS reads a given `bdFrame`, it sets the block size initially to the value passed in `"block.size"`, and then adjusts downward until the block size is no greater than `"max.block.mb"`. Because the default for `"block.size"` is set so high, this effectively ensures that the size of the block is around the given number of megabytes.

The resulting number of rows in a block depends on the types and numbers of columns in the data. Given the default `"max.block.mb"` of 10 megabytes, reading a `bdFrame` with a single numeric column could

be read in blocks of 1,250,000 rows. A bdFrame with 200 numeric columns could be read in blocks of 6,250 rows. The column types also enter into the determination of the number of rows in a block.

Changing Block Size Options

There is rarely a reason to change bd.options("max.block.mb"); however, if you increase it, do not set it to be larger than the physical memory on the computer.

Likewise, there is little need to change bd.options("block.size"). One exception is if you are developing and debugging new code for processing big data. Consider developing code that calls bd.block.apply to processes very large data in a series of chunks. To test whether this code works when the data is broken into multiple blocks, set "block.size" to a very small value, such as bd.options(block.size=10). By following this technique, you can test processing multiple blocks quickly with very small data sets.

Group or Window Blocks

Note that the "block" size determined by these options and the data is distinct from the "blocks" defined in the functions bd.by.group, bd.by.window, bd.split.by.group, and bd.split.by.window. These functions divide their input data into subsets to process as determined by the values in certain columns or a moving window. S-PLUS imposes a limit on the size of the data that can be processed in each block by bd.by.group and bd.by.window: if the number of rows in a block is larger than the block size determined by bd.options("block.size") and bd.options("max.block.mb"), an error is displayed. This limitation does not apply to the functions bd.split.by.group and bd.split.by.window.

To demonstrate this restriction, consider the code below. The variable BIG.GROUPS contains a 1,000-row data.frame with a column GENDER with factor values MALE and FEMALE, split evenly between the rows. If the block size is large enough, we can use bd.by.group to process each of the GENDER groups of 500 rows:

```
BIG.GROUPS <-
  data.frame(GENDER=rep(c("MALE","FEMALE"),
  length=1000), NUM=rnorm(1000))

bd.options(block.size=5000)
```

```
bd.by.group(BIG.GROUPS, by.columns="GENDER",
  FUN=function(df)
  data.frame(GENDER=df$GENDER[1],
  NROW=nrow(df)))

GENDER      NROW
1 FEMALE    500
2 MALE      500
```

If the block size is set below the size of the groups, this same operation will generate an error:

```
bd.options(block.size=10)

bd.by.group(BIG.GROUPS, by.columns="GENDER",
  FUN=function(df)
  data.frame(GENDER=df$GENDER[1],
  NROW=nrow(df)))

Problem in bd.internal.exec.node(engine.class = :
BDLManager$BDLSplusScriptEngineNode (0): Problem in
bd.internal.by.group.script(IM, function(..: can't process
block with 500 rows for group [FEMALE]: can only process 10
rows at a time (check bd.options() values for block.size
and max.block.mb)
Use traceback() to see the call stack
```

In this case, `bd.split.by.group` could be called to divide the data into a list of multiple `bdFrame` objects and process them individually:

```
BIG.GROUPS.LIST <- bd.split.by.group(BIG.GROUPS,
  by.columns="GENDER")

data.frame(GENDER=names(BIG.GROUPS.LIST),
  NROW=sapply(BIG.GROUPS.LIST, nrow, simplify=T),
  row.names=NULL)

GENDER      NROW
1 FEMALE    500
2 MALE      500
```

Another function where block size is a concern is `bd.block.apply`, which applies user-specified S-PLUS code to sequential data blocks. User code called within `bd.block.apply` should not be written to depend on having a particular block size, because the block size is different when the input data has different numbers and types of

columns. When developing such code, test it with several small values of `bd.options("block.size")`, to ensure that it does not depend on the block size.

BIG DATA STRING AND FACTOR ISSUES

Big data columns of types `character` and `factor` have limitations that are not present for regular `data.frame` objects. Most of the time, these limitations do not cause problems, but in some situations, warning messages can appear, indicating that long strings have been truncated, or factors with too many levels had some values changed to `NA`. This section explains why these warnings may appear, and how to deal with them.

String Column Widths

When a `bdFrame` character column is initially defined, before any data is stored in it, the maximum number of characters (or string width) that can appear in the column must be specified. This restriction is necessary for rapid access to the cache file. Once this is specified, an attempt to store a longer string in the column causes the string to be truncated and generate a warning. It is important to specify this maximum string width correctly. All of the big data operations attempt to estimate this width, but there are situations where this estimated value is incorrect. In these cases, it is possible to explicitly specify the column string width.

To retrieve the actual column string widths used in a particular `bdFrame`, call the function `bd.string.column.width`.

Unless the column string width is explicitly specified in other ways, the default string width for newly-created columns is set with the following option. The default value is 32.

```
bd.options("string.column.width")
```

When you convert a `data.frame` with a character column to a `bdFrame`, the maximum string width in the column data is used to set the `bdFrame` column string width, so there is no possibility of string truncation.

String Widths and importData

When you import a big data object using `importData` for file types other than ASCII text, S-PLUS determines the maximum number of characters in each string column and uses this value to set the `bdFrame` column string width.

When you import ASCII text files, S-PLUS measures the maximum number of characters in each column while scanning the file to determine the column types. The number of lines scanned is controlled by the argument `scanLines`. If this is too small, and the scan stops before some very long strings, it is possible for the estimated column width to be too low. For example, the following code generates a file with steadily-longer strings.

```
f <- tempfile()
cat("strsize,str\n",file=f)
for(x in 1:30) {
   str <- paste(rep("abcd:",x),collapse="")
   cat(nchar(str), ",", str, "\n", sep="",
   append=T, file=f)
 }
```

Importing this file with the default `scanLines` value (256) detects that the maximum string has 150 characters, and sets this column string length correctly.

```
dat <- importData(f, type="ASCII", stringsAsFactors=F,
   bigdata=T)
dat
```

```
**bdFrame: 30 rows, 2 columns**
strsize      str
1  5       abcd:
2  10     abcd:abcd:
3  15    abcd:abcd:abcd:
4  20   abcd:abcd:abcd:abcd:
5  25  abcd:abcd:abcd:abcd:abcd:
   ... 25 more rows ...
```

```
bd.string.column.width(dat)
```

```
      strsize       str
      -1           150
```

(In the above output, the `strsize` value of `-1` represents the value for non-character columns.)

If you import this file with the `scanLines` argument set to scan only the first few lines, the column string width is set too low. In this case, the column string width is set to 45 characters, so longer strings are truncated, and a warning is generated:

```
dat <- importData(f, type="ASCII", stringsAsFactors=F,
    bigdata=T, scanLines=10)

Warning messages:
"ReadTextFileEngineNode (0): output column str has 21
string values truncated because they were longer than the
column string width of 45 characters -- maximum string size
before truncation was 150 characters" in:
bd.internal.exec.node(engine.class = engine.class, ...
```

You can read this data correctly without scanning the entire file by explicitly setting `bd.options("default.string.column.width")` before the call to `importData`:

```
bd.options("default.string.column.width"=200)
dat <- importData(f, type="ASCII", stringsAsFactors=F,
    bigdata=T, scanLines=10)
bd.string.column.width(dat)

strsize          str
 -1              200
```

This string truncation does not occur when S-PLUS reads long strings as factors, because there is no limit on factor-level string length.

One more point to remember when you import strings: the low-level `importData` and `exportData` code truncates any strings (either character strings or factor levels) that have more than 254 characters. S-PLUS generates a warning in `importData` if `bigdata=T` if it encounters such strings.

String Widths and `bd.create.columns`

You can use one of the following techniques for setting string column widths explicitly:

- To set the default width (if it is not determined some other way), use `bd.options("string.column.width")`.
- To override the default column string widths, in `bd.block.apply`, specify the `out1.column.string.widths` list element when `IM$test==T`, or when outputting the first non-`NULL` output block.
- To set the width for new output columns, use the `string.column.width` argument to `bd.create.columns`. When you use `bd.create.columns` to create a new character column, you must set the column string width. You can set

this width explicitly with the `string.column.width` argument. If you set it smaller than the maximum string generated, then this will generate a warning:

```
bd.create.columns(as.bdFrame(fuel.frame),
    "Type+Type", "t2", "character",
    string.column.width=6)

Warning in bd.internal.exec.node(engine.class = engi..:
"CreateColumnsEngineNode (0): output column t2 has 53
string values truncated because they were longer than the
column string width of 6 characters -- maximum string size
before truncation was 14 characters"

**bdFrame: 60 rows, 6 columns**
  Weight Disp. Mileage  Fuel  Type  t2
1  2560   97   33   3.030303 Small SmallS
2  2345  114   33   3.030303 Small SmallS
3  1845   81   37   2.702703 Small SmallS
4  2260   91   32   3.125000 Small SmallS
5  2440  113   32   3.125000 Small SmallS
... 55 more rows ...
```

If the character column width is not set with the `string.column.width` argument, the value is estimated differently, depending on whether the `call.splus` argument is true or false. If `row.language=T`, the expression is analyzed to determine the maximum length string that could possibly be generated. This estimate is not perfect, but it works well enough most of the time.

If `row.language=F`, the first time that the S-PLUS expression is evaluated, the string widths are measured, and the new column's string width is set from this value. If future evaluations produce longer strings, they are truncated, and a warning is generated.

Whether `row.language=T` or `F`, the estimated string widths will never be less than the value of `bd.options("default.string.column.width")`.

Factor Column Levels

Because of the way that `bdFrame` factor columns are represented, a factor cannot have an unlimited number of levels. The number of levels is restricted to the value of the option. (The default is 500.)

```
bd.options("max.levels")
```

If you attempt to create a factor with more than this many levels, a warning is generated. For example:

```
dat <- bd.create.columns(data.frame(num=1:2000),
   "'x'+num", "f", "factor")

Warning messages:
"CreateColumnsEngineNode (0): output column f has 1500 NA
values due to categorical level overflow (more than 500
levels) -- you may want to change this column type from
categorical to string" in: bd.internal.ex\
ec.node(engine.class = engine.class, node.props =
node.props, ....

summary(dat)

 num          f
 Min.:  1.0     x99: 1
 1st Qu.: 500.8   x98: 1
 Median: 1001.0   x97: 1
 Mean: 1001.0    x96: 1
 3rd Qu.: 1500.0   x95: 1
 Max.: 2000.0 (Other): 495
      NA's:1500
```

You can increase the `"max.levels"` option up to 65,534, but factors with so many levels should probably be represented as character strings instead.

Note

Strings are used for identifiers (such as street addresses or social security numbers), while factors are used when you have a limited number of categories (such as state names or product types) that are used to group rows for tables, models, or graphs.

String Truncation and Level Overflow Errors

Normally, if strings are truncated or factor levels overflow, S-PLUS displays a warning with detailed information on the number of altered values after the operation is completed. You can set the following options to make an error occur immediately when a string truncation or level overflow occurs.

```
bd.options("error.on.string.truncation"=T)
bd.options("error.on.level.overflow"=T)
```

The default for both options is `F`. If one of these is set to `T`, an error occurs, with a short error message. Because all of the data has not been processed, it is impossible to determine how many values might be effected.

These options are useful in situations where you are performing a lengthy operation, such as importing a huge data set, and you want to terminate it immediately if there is a possible problem.

STORING AND RETRIEVING LARGE S OBJECTS

When you work with very large data, you might encounter a situation where an object or collection of objects is too large to fit into available memory. The Big Data library offers two functions to manage storing and retrieving large data objects:

- `bd.pack.object`
- `bd.unpack.object`

This topic contains examples of using these functions.

Managing Large Amounts of Data

Suppose you want to create a list containing thousands of model objects, and a single list containing all of the models is too large to fit in your available memory. By using the function `bd.pack.object`, you can store each model in an external cache, and create a list of the smaller "packed" models. You can then use `bd.unpack.object` to restore the models to manipulate them.

Creating a Packed Object with `bd.pack.object`

In the following example, use the data object `fuel.frame` to create 1000 linear models. The resulting object takes about 6MB.

In the **Commands** window, type the following:

```
#Create the linear models:
many.models <- lapply(1:1000, function(x)
  lm(Fuel ~ Weight + Disp., sample(fuel.frame, size=30)))

#Get the size of the object:
object.size(many.models)

[1] 6210981
```

You can make a smaller object by packing each model. While this exercise takes longer, the resulting object is smaller than 2MB.

In the **Commands** window, type the following:

```
#Create the packed linear models:
many.models.packed <- lapply(1:1000,
  function(x) bd.pack.object(
  lm(Fuel ~ Weight + Disp., sample(fuel.frame, size=30))))
```

```
#Get the size of the packed object:
object.size(many.models.packed)

[1] 1880041
```

Restoring a Packed Object with `bd.unpack.object`

Remember if you use `bd.pack.object`, you must unpack the object to use it again. The following example code unpacks some of the models within `many.models.packed` object and displays them in a plot.

In the **Commands** window, type the following:

```
for(x in 1:5)
  plot(
  bd.unpack.object(many.models.packed[[x]]),
  which.plots=3)
```

Summary

The above example shows a space difference of only a few MB, (6MB to 2MB), which is probably not a large enough saving to take the time to pack the object. However, if each of the model objects were very large, and the whole list were too large to represent, the packed version would be useful.

INCREASING EFFICIENCY

The Big Data library offers several alternatives to standard S-PLUS functions, to provide greater efficiency when you work with a large data set. Key efficiency functions include:

Table G.1: *Efficient Big Data library functions.*

Function name	Description
`bd.select.rows`	Use to extract specific columns and a block of contiguous rows.
`bd.filter.rows`	Use to keep all rows for which a condition is `TRUE`.
`bd.create.columns`	Use to add columns to a data set.

The following section provides comparisons between these Big Data library functions and their standard S-PLUS function equivalents

`bd.select.rows`

Using `bd.select.rows` to extract a block of rows is much more efficient than using standard subscripting. Some standard subscripting and `bd.select.rows` equivalents include the following:.

Table G.2: *`bd.select.rows` efficiency equivalents.*

Standard S-PLUS subscripting function	**`bd.select.rows` equivalent**
`x[, "Weight"]`	`bd.select.rows(x, columns="Weight")`
`x[1:1000, c(1,3)]`	`bd.select.rows(x, from=1, to=1000, columns=c(1,3))`

`bd.filter.rows`

Using `bd.filter.rows` is equivalent to subscripting rows with a logical vector. By default, `bd.filter.rows` uses an "expression language" that provides quick evaluation of row-oriented expressions. Alternatively, you can use the full range of S-PLUS row functions by

setting the bd.filter.rows argument row.language=F, but the computation is less efficient. Some standard subscripting and bd.filter.rows equivalents include the following:.

Table G.3: *bd.filter.rows efficiency equivalents.*

Standard S-PLUS subscripting function	bd.filter.rows **equivalent**
x[x$Weight > 100,]	bd.filter.rows(x, "Weight > 100")
x[pnorm(x$stat) > 0.5 ,]	bd.filter.rows(x, "pnorm(stat) > 0.5", row.language=F)

bd.create.columns

Like bd.filter.rows, bd.create.columns offers you a choice of using the more efficient expression language or the more flexible general S-PLUS functions. Some standard subscripting and bd.create.columns equivalents include the following:

Table G.4: *bd.create.columns efficiency equivalents.*

Standard S-PLUS subscripting function	bd.create.columns **equivalent**
x$d <- (x$a+x$b)/x$c	x <- bd.create.columns(x, "(a+b)/c", "d")
x$pval <- pnorm(x$stat)	x <- bd.create.columns(x, "pnorm(stat)", "pval", row.language=F)
y <- (x$a+x$b)/x$c	y <- bd.create.columns(x, "(a+b)/c", "d", copy=F)

Note that in the last function, above, specifying copy=F creates a new column without copying the old columns.

APPENDIX: BIG DATA LIBRARY FUNCTIONS

Introduction **202**

Big Data Library Functions **203**
- Data Import and Export 203
- Object Creation 204
- Big Vector Generation 205
- Big Data Library Functions 206
- Data Frame and Vector Functions 214
- Graph Functions 228
- Data Modeling 230
- Time Date and Series Functions 234

INTRODUCTION

The Big Data library is supported by many standard S-PLUS functions, such as basic statistical and mathematical functions, properties functions, densities and quantiles functions, and so on. For more information about these functions, see their individual help topics. (To display a function's help topic, in the **Commands** window, type `help(`*`functionname`*`)`.)

The Big Data library also contains functions specific to big data objects. These functions include the following.

- Import and export functions.
- Object creation functions
- Big vector generating functions.
- Data exploration and manipulation functions.
- Traditional and Trellis graphics functions.
- Modeling functions.

These functions are described further in the following section.

BIG DATA LIBRARY FUNCTIONS

The following tables list the functions that are implemented in the Big Data library.

Data Import and Export

For more information and usage examples, see the functions' individual help topics.

Table A.1: *Import and export functions.*

Function name	Description
`data.dump`	Creates a file containing an ASCII representation of the objects that are named.
`data.restore`	Puts data objects that had previously been put into a file with `data.dump` into the specified database.
`exportData`	Exports a `bdFrame` to the specified file or database format. Not all standard S-PLUS arguments are available when you import a large data set. See `exportData` in the S-PLUS Language Reference for more information.
`importData`	When you set the `bigdata` flag to `TRUE`, imports data from a file or database into a `bdFrame`. Not all standard S-PLUS arguments are available when you import a large data set. See `importData` in the S-PLUS Language Reference for more information.

Object Creation

The following methods create an object of the specified type. For more information and usage examples, see the functions' individual help topics.

Table A.2: *Big Data library object creation functions*

Function
`bdCharacter`
`bdCluster`
`bdFactor`
`bdFrame`
`bdGlm`
`bdLm`
`bdLogical`
`bdNumeric`
`bdPrincomp`
`bdSignalSeries`
`bdTimeDate`
`bdTimeSeries`
`bdTimeSpan`

Big Vector Generation

For the following methods, set the `bigdata` argument to `TRUE` to generate a `bdVector`. This instruction applies to all functions in this table. For more information and usage examples, see the functions' individual help topics.

Table A.3: *Vector generation methods for large data sets.*

Method name
`rbeta`
`rbinom`
`rcauchy`
`rchisq`
`rep`
`rexp`
`rf`
`rgamma`
`rgeom`
`rhyper`
`rlnorm`
`rlogis`
`rmvnorm`
`rnbinom`
`rnorm`

Table A.3: *Vector generation methods for large data sets. (Continued)*

Method name
`rnrange`
`rpois`
`rstab`
`rt`
`runif`
`rweibull`
`rwilcox`

Big Data Library Functions

The Big Data library introduces a new set of `"bd"` functions designed to work efficiently on large data. For best performance, it is important that you write code minimizing the number of passes through the data. The Big Data library functions minimize the number of passes made through the data. Use these functions for the best performance. For more information and usage examples, see the functions' individual help topics.

Data Exploration Functions

Table A.4: *Data exploration functions.*

Function name	Description
`bd.cor`	Computes correlation or covariances for a data set. In addition, computes correlations or covariances between a single column and all other columns, rather than computing the full correlation/covariance matrix.
`bd.crosstabs`	Produces a series of tables containing counts for all combinations of the levels in categorical variables.
`bd.data.viewer`	Displays the data viewer window, which displays the input data in a scrollable window, as well as information about the data columns (names, types, means, and so on).
`bd.univariate`	Computes a wide variety of univariate statistics. It computes most of the statistics returned by `PROC UNIVARIATE` in SAS.

Data Manipulation Functions

Table A.5: *Data manipulation functions.*

Function name	Description
`bd.aggregate`	Divides a data object into blocks according to the values of one or more columns, and then applies aggregation functions to columns within each block.
`bd.append`	Appends one data set to a second data set.
`bd.bin`	Creates new categorical variables from continuous variables by splitting the numeric values into a number of bins. For example, it can be used to include a continuous age column as ranges (<18, 18-24, 25-35, and so on).
`bd.block.apply`	Executes an S-PLUS script on blocks of data, with options for reading multiple input datasets and generating multiple output data sets, and processing blocks in different orders.
`bd.by.group`	Apply an arbitrary S-PLUS function to multiple data blocks within the input dataset.
`bd.by.window`	Apply an arbitrary S-PLUS function to multiple data blocks defined by a moving window over the input dataset.
`bd.coerce`	Converts an object from a standard data frame to a `bdFrame`, or vice versa.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.create.columns`	Creates columns based on expressions.
`bd.duplicated`	Determine which rows in a dataset are unique.
`bd.filter.columns`	Removes one or more columns from a data set.
`bd.filter.rows`	Filters rows that satisfy the specified expression.
`bd.join`	Creates a composite data set from two or more data sets. For each data set, specify a set of key columns that defines the rows to combine in the output. Also, for each data set, specify whether to output unmatched rows.
`bd.modify.columns`	Changes column names or types. Can also be used to drop columns.
`bd.normalize`	Centers and scales continuous variables. Typically, variables are normalized so that they follow a standard Gaussian distribution (means of 0 and standard deviations of 1). To do this, `bd.normalize` subtracts the mean or median, and then divides by either the range or standard deviation.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.partition`	Randomly samples the rows of your data set to partition it into three subsets for training, testing, and validating your models.
`bd.relational.difference`	Get differing rows from two input data sets.
`bd.relational.divide`	Given a `Value` column and a `Group` column, determine which values belong to a given Membership as defined by a set of `Group` values.
`bd.relational.intersection`	Join two input data sets, ignoring all unmatched columns, with the common columns acting as key columns.
`bd.relational.join`	Join two input data sets with the common columns acting as key columns.
`bd.relational.product`	Join two input data sets, ignoring all matched columns, by performing the cross product of each row.
`bd.relational.project`	Remove one or more columns from a data set.
`bd.relational.restrict`	Select the rows that satisfy an expression. Determines whether each row should be selected by evaluating the restriction. The result should be a logical value.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.relational.union`	Retrieve the relational union of two data sets. Takes two inputs (`bdFrame` or `data.frame`). The output contains the common columns and includes the rows from both inputs, with duplicate rows eliminated.
`bd.remove.missing`	Drops rows with missing values, or replaces missing values with the column mean, a constant, or values generated from an empirical distribution, based on the observed values.
`bd.reorder.columns`	Changes the order of the columns in the data set.
`bd.sample`	Samples rows from a dataset, using one of several methods.
`bd.select.rows`	Extracts a block of data, as specified by a set of columns, start row, and end row.
`bd.shuffle`	Randomly shuffles the rows of your data set, reordering the values in each of the columns as a result
`bd.sort`	Sorts the data set rows, according to the values of one or more columns.
`bd.split`	Splits a data set into two data sets according to whether each row satisfies an expression.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.sql`	Specifies data manipulation operations using SQL syntax. • The `Select`, `Insert`, `Delete`, and `Update` statements are supported. • The column identifiers are case sensitive. • SQL interprets periods in names as indicating fields within tables; therefore, column names should not contain periods if you plan to use `bd.sql`. • Mathematical functions are allowed for aggregation (avg, min, max, sum, count, stdev, var). The following functionality is not implemented: • distinct • mathematical functions in set or select, such as `abs`, `round`, `floor`, and so on. • natural join • union • merge • between • subqueries
`bd.stack`	Combines or stacks separate columns of a data set into a single column, replicating values in other columns as necessary.

Table A.5: *Data manipulation functions. (Continued)*

Function name	Description
`bd.string.column.width`	Returns the maximum number of characters that can be stored in a big data string column.
`bd.transpose`	Turns a set of columns into a set of rows.
`bd.unique`	Remove all duplicated rows from the dataset so that each row is guaranteed to be unique.
`bd.unstack`	Separates one column into a number of columns based on a grouping column.

Programming

Table A.6: *Programming functions.*

Function name	Description
`bd.cache.cleanup`	Cleans up cache files that have not been deleted by the garbage collection system. (This is most likely to occur if the entire system crashes.)
`bd.cache.info`	Analyzes a directory containing big data cache files and returns information about cache files, references counts, and unknown files.
`bd.options`	Controls S-PLUS options used when processing big data objects.
`bd.pack.object`	Packs any object into an external cache.

Table A.6: *Programming functions. (Continued)*

Function name	Description
`bd.split.by.group`	Divide a dataset into multiple data blocks, and return a list of these data blocks.
`bd.split.by.window`	Divide a dataset into multiple data blocks, defined by a moving window over the dataset, and return a list of these data blocks.
`bd.unpack.object`	Unpacks a `bdPackedObject` object that was previously stored in the cache using `bd.pack.object`.

Data Frame and Vector Functions

The following table lists the functions for both data frames (`bdFrame`) and vectors (`bdVector`). The the cross-hatch (#) indicates that the function is implemented for the corresponding object type. The Comment column provides information about the function, or indicates which `bdVector`-derived class(es) the function applies to. For more information and usage examples, see the functions' individual help topics.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`.*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`-`	#	#	
`!=`	#	#	
`$`		#	
`$<-`		#	
`[`	#	#	
`[[`	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`[[<-`	#	#	
`[<-`	#	#	
`abs`	#		
`aggregate`	#	#	
`all`	#	#	
`all.equal`	#	#	
`any`	#	#	
`anyMissing`	#	#	
`append`	#		
`apply`		#	
`Arith`	#	#	
`as.bdCharacter`	#		
`as.bdFactor`	#		
`as.bdFrame`	#	#	
`as.bdLogical`	#		Handles all `bdVector`-derived object types.
`as.bdVector`	#	#	
`attr`	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`attr<-`	#	#	
`attributes`	#	#	
`attributes<-`	#	#	
`bdFrame`	#	#	Constructor. Inputs can be `bdVectors`, `bdFrames`, or ordinary objects.
`boxplot`	#	#	Handles `bdNumeric`.
`by`		#	
`casefold`	#		
`ceiling`	#		
`coerce`	#	#	
`colIds`		#	
`colIds<-`		#	
`colMaxs`	#	#	
`colMeans`	#	#	
`colMins`	#	#	
`colRanges`	#	#	
`colSums`	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`colVars`	#	#	
`concat.two`	#	#	
`cor`	#	#	
`cut`	#		
`dbeta`	#		Density, cumulative distribution (CDF), and quantile function.
`dbinom`	#		Density, CDF, and quantile function.
`dcauchy`	#		Density, CDF, and quantile function.
`dchisq`	#		Density, CDF, and quantile function.
`density`	#		
`densityplot`		#	
`dexp`	#		Density, CDF, and quantile function.
`df`	#		Density, CDF, and quantile function.
`dgamma`	#		Density, CDF, and quantile function.
`dgeom`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`dhyper`	#		Density, CDF, and quantile function.
`diff`	#	#	
`digamma`	#		
`dim`		#	
`dimnames`		#	a `bdFrame` has no row names.
`dimnames<-`		#	a `bdFrame` has no row names.
`dlnorm`	#		Density, CDF, and quantile function.
`dlogis`	#		Density, CDF, and quantile function.
`dmvnorm`		#	Density and CDF function.
`dnbinom`	#		Density, CDF, and quantile function.
`dnorm`	#		Density, CDF, and quantile function.
`dnrange`	#		Density, CDF, and quantile function.
`dpois`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`dt`	#		Density, CDF, and quantile function.
`dunif`	#		Density, CDF, and quantile function.
`duplicated`	#	#	Density, CDF, and quantile function.
`durbinWatson`	#		Density, CDF, and quantile function.
`dweibull`	#		Density, CDF, and quantile function.
`dwilcox`	#		Density, CDF, and quantile function.
`floor`	#	#	
`format`	#	#	
`formula`		#	
`grep`	#		
`hist`	#		
`hist2d`	#		
`histogram`		#	
`html.table`	#	#	
`intersect`	#		

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`is.all.white`	#		
`is.element`	#		
`is.finite`	#	#	
`is.infinite`	#	#	
`is.na`	#	#	
`is.nan`	#	#	
`is.number`	#	#	
`is.rectangular`	#	#	
`kurtosis`	#		Handles `bdNumeric`.
`length`	#	#	
`levels`	#		Handles `bdFactor`.
`levels<-`	#		Handles `bdFactor`.
`mad`	#		
`match`	#	#	
`Math`	#	#	Operand function.
`Math2`	#	#	Operand function.
`matrix`	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`mean`	#	#	
`median`	#		
`merge`	#	#	
`na.exclude`	#	#	
`na.omit`	#	#	
`names`	#	#	`bdVector` cannot have names.
`names<-`	#	#	`bdVector` cannot have names.
`nchar`	#		Handles `bdCharacter`, not `bdFactor`.
`ncol`		#	
`notSorted`	#		
`nrow`		#	
`numberMissing`	#	#	
`Ops`	#	#	
`pairs`		#	
`pbeta`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`pbinom`	#		Density, CDF, and quantile function.
`pcauchy`	#		Density, CDF, and quantile function.
`pchisq`	#		Density, CDF, and quantile function.
`pexp`	#		Density, CDF, and quantile function.
`pf`	#		Density, CDF, and quantile function.
`pgamma`	#		Density, CDF, and quantile function.
`pgeom`	#		Density, CDF, and quantile function.
`phyper`	#		Density, CDF, and quantile function.
`plnorm`	#		Density, CDF, and quantile function.
`plogis`	#		Density, CDF, and quantile function.
`plot`	#	#	
`pmatch`	#		
`pmvnorm`		#	Density and CDF function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`pnbinom`	#		Density, CDF, and quantile function.
`pnorm`	#		Density, CDF, and quantile function.
`pnrange`	#		Density, CDF, and quantile function.
`ppois`	#		Density, CDF, and quantile function.
`print`	#	#	
`pt`	#		Density, CDF, and quantile function.
`punif`	#		Density, CDF, and quantile function.
`pweibull`	#		Density, CDF, and quantile function.
`pwilcox`	#		Density, CDF, and quantile function.
`qbeta`	#		Density, CDF, and quantile function.
`qbinom`	#		Density, CDF, and quantile function.
`qcauchy`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for* `bdVector` *and* `bdFrame`*. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`qchisq`	#		Density, CDF, and quantile function.
`qexp`	#		Density, CDF, and quantile function.
`qf`	#		Density, CDF, and quantile function.
`qgamma`	#		Density, CDF, and quantile function.
`qgeom`	#		Density, CDF, and quantile function.
`qhyper`	#		Density, CDF, and quantile function.
`qlnorm`	#		Density, CDF, and quantile function.
`qlogis`	#		Density, CDF, and quantile function.
`qnbinom`	#		Density, CDF, and quantile function.
`qnorm`	#		Density, CDF, and quantile function.
`qnrange`	#		Density, CDF, and quantile function.
`qpois`	#		Density, CDF, and quantile function.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`qq`		#	
`qqmath`		#	
`qqnorm`	#		
`qqplot`	#		
`qt`	#		Density, CDF, and quantile function.
`quantile`	#		
`qunif`	#		Density, CDF, and quantile function.
`qweibull`	#		Density, CDF, and quantile function.
`qwilcox`	#		Density, CDF, and quantile function.
`range`	#		
`rank`	#		
`replace`	#		
`rev`	#	#	
`rle`	#		
`row.names`		#	Always `NULL`.
`row.names<-`		#	Does nothing.

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`rowIds`		#	Always `NULL`.
`rowIds<-`		#	Does nothing.
`rowMaxs`		#	
`rowMeans`		#	
`rowMins`		#	
`rowRanges`		#	
`rowSums`		#	
`rowVars`		#	
`runif`	#		
`sample`	#	#	
`scale`		#	
`setdiff`	#		
`shiftPositions`	#		
`show`	#	#	
`skewness`	#		Handles `bdNumeric`.
`sort`	#		
`split`		#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	`bdVector`	`bdFrame`	`Optional Comment`
`stdev`	#		Handles `bdCharacter`.
`sub`	#	#	
`sub<-`		#	
`substring`	#		
`substring<-`	#		
`Summary`	#	#	Operand function.
`summary`	#	#	
`sweep`		#	
`t`		#	
`tabulate`	#		Handles `bdNumeric`.
`tapply`	#	#	
`trigamma`	#		
`union`	#		
`unique`	#	#	
`var`	#	#	
`which.infinite`	#	#	
`which.na`	#	#	

Table A.7: *Functions implemented for `bdVector` and `bdFrame`. (Continued)*

Function Name	bdVector	bdFrame	Optional Comment
which.nan	#	#	
xy2cell	#		
xyCall	#		
xyplot		#	

Graph Functions

For more information and examples for using the traditional graph functions, see their individual help topics, or see Chapter 5, Creating Graphical Displays of Large Data Sets.

Table A.8: *Traditional graph functions.*

Function name
barplot
boxplot
contour
dotchart
hexbin
hist
hist
hist2d
image
interp

Table A.8: *Traditional graph functions. (Continued)*

Function name
pairs
persp
pie
plot
qqnorm
qqplot

For more information about using the Trellis graph functions, see their individual help topics, or see Chapter 5, Creating Graphical Displays of Large Data Sets.

Table A.9: *Trellis graph functions.*

Function name
barchart
contourplot
densityplot
dotplot
histogram
levelplot
piechart
qq

Note
The `cloud` and `parallel` graphics functions are not implemented for `bdFrames`.

Data Modeling

For more information and usage examples, see the functions' individual help topics.

Table A.10: *Fitting functions*

Function name
`bdCluster`
`bdGlm`
`bdLm`
`bdPrincomp`

Table A.11: *Other modeling utilities.*

Function name
`bd.model.frame.and.matrix`
`bs`
`ns`
`spline.des`
`C`
`contrasts`
`contrasts<-`

Model Methods The following table identifies functions implemented for generalized linear modeling, linear regression, principal components modeling, and clustering. The cross-hatch (#) indicates the function is implemented for the corresponding modeling type.

Table A.12: *Modeling and Clustering Functions.*

Function name	**Generalized linear modeling** (`bdGlm`)	**Linear Regression** (`bdLm`)	**principal components** (`bdPrincomp`)	`bdCluster`
`AIC`		#		
`all.equal`		#		
`anova`	#	#		
`BIC`		#		
`coef`	#	#		
`deviance`	#	#		
`durbinWatson`		#		
`effects`		#		
`family`	#	#		
`fitted`	#	#	#	#
`formula`	#	#		
`kappa`		#		
`labels`		#		
`loadings`			#	

Table A.12: *Modeling and Clustering Functions. (Continued)*

Function name	**Generalized linear modeling** (`bdGlm`)	**Linear Regression** (`bdLm`)	**principal components** (`bdPrincomp`)	`bdCluster`
`logLik`		#		
`model.frame`		#		
`model.matrix`		#		
`plot`		#	#	
`predict`	#	#	#	#
`print`	#	#	#	#
`print.summary`	#	#	#	
`qqnorm`		#		
`residuals`	#	#		
`screeplot`			#	
`step`	#	#		
`summary`	#	#	#	

Predict from Small Data Models

This table lists the small data models that support the `predict` function. For more information and usage examples, see the functions' individual help topics.

Table A.13: *Predicting from small data models.*

Small data model using `predict` function
`arima.mle`
`bs`
`censorReg`
`coxph`
`coxph.penal`
`discrim`
`factanal`
`gam`
`glm`
`gls`
`gnls`
`lm`
`lme`
`lmList`
`lmRobMM`
`loess`
`loess.smooth`

Table A.13: *Predicting from small data models. (Continued)*

Small data model using `predict` **function**
`mlm`
`nlme`
`nls`
`ns`
`princomp`
`safe.predict.gam`
`smooth.spline`
`smooth.spline.fit`
`survreg`
`survReg`
`survReg.penal`
`tree`

Time Date and Series Functions

The following tables include time date creation functions and functions for manipulating time and date, time span, time series, and signal series objects.

Time Date Creation

Table A.14: *Time date creation functions.*

Function name	Description
`bdTimeDate`	The object constructor. Note that when you call the `timeDate` function with any big data arguments, then a `bdTimeDate` object is created.
`timeCalendar`	Standard S-PLUS function. When you call the `timeCalendar` function with any big data arguments, then a `bdTimeDate` object is created
`timeSeq`	Standard S-PLUS function; to use with a large data set, set the `bigdata` argument to `TRUE`.

In the following table, the cross-hatch (#) indicates that the function is implemented for the corresponding class. If the table cell is blank, the function is not implemented for the class. This list includes `bdVector` objects (`bdTimeDate` and `bdTimeSpan`) and `bdSeries` classes (`bdSignalSeries`, `bdTimeSeries`).

Table A.15: *Time Date and Series Functions.*

Function	`bdTimeDate`	`bdTimeSpan`	`bdSeries`	`bdSignalSeries`	`bdTimeSeries`
`-`	#	#			
`[`		#	#		
`[<-`		#			
`+`	#	#			
`align`			#		

Table A.15: *Time Date and Series Functions. (Continued)*

Function	bdTimeDate	bdTimeSpan	bdSeries	bdSignalSeries	bdTimeSeries
all.equal	#	#			
Arith	#	#			
as.bdFrame					#
as.bdLogical	#				
bd.coerce		#			
ceiling	#	#			
coerce	#	#	#		
cor	#	#			
cumsum		#			
cut	#	#			
data.frameAux					#
days	#				
deltat					#
diff			#		
end			#		
floor	#	#			
hms	#				

Table A.15: *Time Date and Series Functions. (Continued)*

Function	bdTimeDate	bdTimeSpan	bdSeries	bdSignalSeries	bdTimeSeries
hours	#				
match	#	#			
Math	#	#			
Math2	#	#			
max	#	#			
mdy	#				
mean	#	#			
median	#	#			
min	#	#			
minutes	#				
months	#				
plot	#			#	#
quantile	#	#			
quarters	#				
range	#	#			
seconds	#				
seriesLag			#		

Table A.15: *Time Date and Series Functions. (Continued)*

Function	bdTimeDate	bdTimeSpan	bdSeries	bdSignalSeries	bdTimeSeries
shiftPositions	#				
show			#		
sort	#	#			
sort.list	#	#			
split			#		
start			#		
substring<-	#	#	#	#	#
sum		#			
Summary	#	#			
summary	#	#			
timeConvert	#				
trunc	#	#			
var	#	#			
wdydy	#				
weekdays	#				
yeardays	#				
years	#				

INDEX

Symbols

.Data database 16
.metadata database 16

Numerics

64-bit systems 64

A

add a task
    in script file 52
anonymous functions
    displaying 21
anova 76

B

background color
    console 19
    Workbench script editor 21
basic algebra 82
bd.create.columns 101
bd.options 74
bdCharacter 75
bdCluster 73, 76, 173
bdFactor 75
bdFrame 69, 73, 77
    introducing the new data type 62
bdGLM 73
bdGlm 76
bdLM 73
bdLm 76
bdLogical 75
bdNumeric 75
bdPrincomp 73, 76
bdSeries 73
    data 77
    positions 77
    units 77
bdSignalSeries 73
bdTimeDate 75, 81
bdTimeSeries 73
bdTimeSpan 75
bdVector 73, 74, 78
Boston housing example 163
build a model 7

C

changing databases
    adding a directory 47
    adding a library 46
    adding a module 47
classes
    bdCharacter 77
    bdCluster 77
    bdFactor 77
    bdGlm 77
    bdLm 77
    bdLogical 77
    bdNumeric 77
    bdPrincomp 77
    bdSignalSeries 77
    bdTimeDate 77

bdTimeSeries 77
bdTimeSpan 77
bdVector 77
coef 76
Commands window 2
comparing versions 56
components
Principal Components 170
console options 19
Console View 26, 28
copying from script to console 15
create a Workbench project 41
Creating 109
crossprod 82
custom color
setting 19, 20, 21

D

data frame 73
data streaming 62
debugging 15
drop NA values 109

E

Eclipse 11
edit code 49
empty project
creating 41
evaluating expressions 40
existing files
creating a project for 41
existing project
importing files for 42
exporting data 104
external files
opening 23, 57

F

file associations 19
filtering files 57
fitted 76
format code 40
formula 76
function help 14
functions to watch 21

G

generalized linear modeling 162
graphical display 7
graphical user interface 4
graphics functions 78
GUI support
Big Data library 2, 4

H

help, displaying 38
History View 26, 29, 53

I

import data 6
importing data 6
Boston housing example 181
importing multiple files 107
Stock example 107
importing files 42

J

join columns 103

K

K-means 173

L

linear model 163
linear regression 162
Boston Housing example 163
line numbers 49, 50
displaying 21, 38
loading Big Data library by default 2

M

manipulate data 6
metadata 63
model 75
modeling functions 79
multiple projects 43

N

Navigator View 27, 48, 54
New Project wizard 23

O

Objects View 26, 30
opening external files 39
Outline View 26, 31
out-of-memory
    data storage 4
    processing 61
Output View 26, 33

P

Perspective 12
perspective 24
    preferences 18
plot 76
predict 76
    basis matrix for polynomial splines 180
    censored data 180
    factor analysis model 180
    linear mixed-effects models 180
    local regression model 181
    nonlinear mixed-effects model 181
    nonlinear regression model 181
    normal linear discriminant function 180
    principal components 181
preferences
    setting 44
Prim4 principal pomponents example 170
Principal Components
    component 170
principal components
    loadings 170
    predict 170
    print 170
    screeplot 170
    summary 170
Problems View 27, 34, 54
project files
    removing 48

R

refreshing
    Objects View 31
    Problems View 34
    Search Path View 35
    views 47
removing
    project files 48
residuals 76
restoring files 56
running code 39, 52
    on startup 20
running scripts 14

S

scalable algorithms 62, 63
script
    creating 48
Script window 2
searching terms 57
Search Path View 27
setting bigdata=T 65
signalSeries 76
simultaneous sessions 11
S-Plus Workbench 11
starting the Workbench 16
stringsAsFactors 65
summary 74, 76

T

task levels 36
task options 23
Tasks View 27
timeDate
    positions 76
timeSeries 76
time series
    creating 110
time series object, creating 109
toggling comment 40

U

units 76

V

vectors 74
view
    customize 45
views
    changing display 46
virtual memory limitations 61

W

Workbench Project 13
Workbench project
    creating 41
Workbench Script Editor 13
Workbench User Guide 13
Workbench View 13
Workspace 12
workspace 16, 18
    changing 18